VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Algoritmy pro převod regulárních výrazů na konečné automaty

Algorithms for transformation of regular expressions to finite automata

2017 Tomáš Vacho

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Tomáš Vacho**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Algoritmy pro převod regulárních výrazů na konečné automaty
Algorithms for Transformation of Regular Expressions to Finite Automata

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je implementovat a porovnat několik různých algoritmů pro převod regulárních výrazů na konečné automaty. Takové algoritmy mají význam zejména v nástrojích používaných pro implementaci lexikální analýzy a v nástrojích pro vyhledávání v textu.

1. Nastudujte několik různých algoritmů pro převod regulárních výrazů na konečné automaty z článků uvedených v seznamu literatury (a případně dalších článků určených vedoucím práce).
2. Implementujte tyto algoritmy (a případně také jejich různé varianty).
3. Otestujte korektnost implementace těchto algoritmů na vhodně zvolených testovacích příkladech.
4. Porovnejte jednotlivé algoritmy z různých hledisek - z hlediska časové a paměťové náročnosti, z hlediska velikosti vytvořených automatů, apod.

Seznam doporučené odborné literatury:

[1] G. Berry, R. Sethi - From Regular Expressions To Deterministic Automata, Theoretical Computer Science 48, pp. 117-126, 1986.
[2] A. Brüggemann-Klein - Regular Expressions into Finite Automata,Theoretical Computer Science 120, pp. 197-213, 1993.
[3] J. Hromkovič, S. Seibert, T. Wilke - Translating Regular Expressions into Small epsilon-Free Nondeterministic Finite Automata, Journal of Computer and System Sciences, Volume 62, Issue 4, pp. 565-588, 2001.
[4] G. Schnitger - Regular Expressions and NFAs without epsilon-Transitions, Proc. of STACS 2006, LNCS 3884, pp. 432-443, 2006.

Další literatura podle pokynů vedoucího práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Zdeněk Sawa, Ph.D.**

Datum zadání: 01.09.2015

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2017 .....

Děkuji vedoucímu práce za odborné vedení, za pomoc a rady při zpracování této práce.

Abstrakt

Tato práce řeší převod regulárních výrazů na konečné automaty. Cílem této práce je naimplementovat několik známých algoritmů v Javě. Zaměřil jsem se na způsob převodu takový, aby bylo v konečném důsledku vždy dosaženo stejného minimálního konečného automatu, ať již je převod realizován libovolným algoritmem. Toto mi umožnilo jednotlivé algoritmy mezi sebou porovnat z hlediska výkonnosti.

Klíčová slova: regulární výrazy, konečné automaty, nedeterministický konečný automat, deterministický konečný automat

Abstract

This work addresses the conversion of regular expressions to finite automata. The aim of this work is to implement several well-known algorithms in Java. I focused on the methods of transfer such that it ultimately always reach the same minimised finite automata, whether the transfer is made by an arbitrary algorithm. This allowed me to different algorithms compared to each other in terms of performance.

Key Words: regular expressions, finite automata, nondeterministic finite automaton, deterministic finite automaton

Obsah

Seznam použitých zkratek a symbolů

CFS	–	common follow sets
DKA	–	deterministický konečný automat
GUI	–	graphic user interface
KA	–	konečný automat
NKA	–	nedeterministický konečný automat
PC	–	personal computer
RV	–	regulární výraz
SNF	–	star normal form
ZNKA	–	zobečněný nedeterministický automat

Seznam obrázků

1 Úvod

Regulární výrazy jsou používány ke specifikaci regulárních jazyků. Konečné automaty jsou používány k rozpoznávání regulárních jazyků. Mnoho počítačových aplikací jako kompilátory, nástroje operačních systémů, či textové editory využívají právě regulárních jazyků. V těchto aplikacích jsou regulární výrazy a konečné automaty používány k rozpoznávání tohoto jazyka.

1.1 Cíl diplomové práce

Cílem práce je implementovat a porovnat několik různých algoritmů pro převod regulárních výrazů na konečné automaty.

1.2 Motivace

Základním algoritmem jsem zvolil **Thompsonův algoritmus**. Ze zadané literatury jsem vybral několik algoritmů, které naimpementuji a porovnám je jak se základním algoritmem, tak mezi sebou.

Jedná se o algoritmy Derivatives, Distinct a Fast z článku From regular expressions to deterministic automata [1], dále o modifikovaný Glushkov algoritmus uvedený v článku Regular expressions into finite automata [2] a konečně o CFS algoritmus uvedený v článku Translating regular expressions into small ε-free nondeterministic finite automata [4].

Algoritmy jsou v literatuře dobře popsány, nyní je tedy zbývá naimplementovat a poté porovnat. Tento stav je tedy východzím bodem pro vypracování této práce.

1.3 Obsah diplomové práce

Obsahem této diplomové práce je implementace aplikace, která prostřednictvím vybraných algoritmů převádí regulární výrazy na konečné automaty.

Aplikace na vstupu obdrží regulární výraz, který rozparsuje do syntaktického stromu. Tento strom je pak vstupem pro jednotlivé algoritmy převodu. Výstupem algoritmu může být jak DKA, tak i (Z)NKA. Tento výstup je pak, je-li to potřeba, převeden do DKA a nakonec i minimalizován.

Výsledek převodu na konečný automat může být prezentován jako grafický výstup (stejně tak výsledky dalších operací jako je například výše uvedená minimalizace), nebo jsou výsledné automaty ukládany v XML. Aplikace umožňuje zadat manuálně jeden regulární výraz, jehož výsledný automat je graficky zobrazen na obrazovce, nebo lze regulární výrazy zadat hromadně ve vstupním TXT souboru. Zadáním hromadného převodu dojde k převedení všech regulárních výrazů ze vstupního TXT souboru prostřednictvím každého z naimplementovaných algoritmů, přičemž jsou při jednotlivých převodech zaznamenávány statistiky, na základě kterých jsou pak algoritmy mezi sebou porovnány.

Aplikace v sobě taktéž zahrnuje generátor regulárních výrazů, který na základě zadaných parametrů generuje regulární výrazy do TXT souboru, který pak může být použit jako vstup pro hromadný převod. Aplikace též obsahuje GUI.

Výsledky hromadných převodů, respektive výsledné statistiky z těchto převodů, jsou pak prezentovány a porovnány v kapitole 5.

Výsledkem práce je tedy aplikace umožňující převádět regulární výrazy na konečné automaty prostřednictvím vybraných algoritmů, přičemž jsou tyto algoritmy mezi sebou navzájem porovnány z hlediska časových nároků.

1.4 Struktura práce

V Kapitole 2 se zaobírám teoretickými aspekty nastudovanými z dostupných zdrojů, které se sebou převádění regulárních výrazů přináší. Je zde popsáno, co to vlastně regulární výraz je, co je to konečný automat a jaké jsou jeho varianty. Jsou zde vyjmenovány všeobecně známé algoritmické postupy aplikovatelné na konečné automaty, které při realizaci jednotlivých převodů budeme potřebovat. Je zde také popsáno, v jaké datové reprezentaci potřebujeme regulární výraz mít, abychom s ním mohli vůbec pracovat.

Kapitola 3 je vyhrazena pro popis samotných algoritmů pro převod regulárních výrazů na konečné automaty.

V Kapitole 4 je popsána vnitřní organizace implementované aplikace, která popsané algoritmy pro převod regulárních výrazů na konečné automaty obsahuje. Jedná se o popis toho, jak aplikace pracuje. Je zde také ukázka uživatelského prostředí. Kompletní programátorská dokumentace implementace se nachází v Příloze A.

Kapitola 5 pak popisuje jak byly jednotlivé algoritmy testovány a jakých výsledků bylo dosaženo. Výsledky jsou pak shrnuty a vyhodnoceny.

Samostatnou částí této práce je pak implementace aplikace demonstrující převod regulárního výrazu zadaného v textové podobě na různé druhy konečných automatů. Kompletní zdrojové kódy, včetně programátorské a uživatelské dokumentace, jsou umístěny v Příloze A.

2 Základní pojmy

V této práci je vycházeno za základních pojmů uvedených v publikaci Introduction to the Theory of Computation [3].

Vždy potřebujeme nejprve vědět s jakou abecedou pracujeme. Abeceda je libovolná konečná množina Σ, která obsahuje písmena abecedy. Posloupnoust symbolů nazýváme slovem; slovem nad abecedou Σ je myšlena jakákoliv konečná posoupnost znaků složená pouze z písmen dané abecedy. Každé slovo má nějakou délku, což je počet znaků daného slova. Prázné slovo, tedy slovo, které neobsahuje žádný znak abecedy, je také slovem. Označujeme ho jako ε a má nulovou délku. Jazyk nad danou abecedou Σ znamená, že jde o jakokouliv podmnožinu slov abecedy Σ^*, tedy množinu všech slov nad abecedou Σ.

S jazyky je také možno provádět jazykové operace. V první řadě to jsou množinové operace, tedy sjednocení, průnik a rozdíl. Dále jsou zde operace zřetězení (spojování znaků či slov do nové posloupnosti) a iterace (rekurentní zřetězení jednoho slova či znaku).

2.1 Regulární výrazy

Regulární výraz je v podstatě vzor pro nalezení a jednoznačné určení textového řetězce. To je využíváno jak v programovacích jazycích, jak již bylo naznačeno v úvodu, tak v běžně používaných nástrojích, jako jsou textové editory.

V běžné praxi se nejčastěji využívají nejrozšířenější Regexp výrazy, nicméně v této práci jsou používány pouze teoretické regulární výrazy, které jsou vhodné pro přesné popsání jazyka, který je přijímán konečným automatem.

Definice 1 *Regulárními výrazy nad abecedou Σ rozumíme množinu RV(Σ) slov v abecedě $\Sigma \cup \{\emptyset, \varepsilon, +, \cdot, *, (,)\}$, přičemž je předpoklad, že $\emptyset, \varepsilon, +, \cdot, *, (,) \notin \Sigma$), a množina zároveň splňuje:*

- *Znaky $\emptyset$, ε a $x \in$ RV(Σ) pro každé písmeno $x \in \Sigma$.*
- *Pokud $\alpha, \beta \in$ RV(Σ), potom i $(\alpha + \beta) \in$ RV(Σ), $(\alpha \cdot \beta) \in$ RV(Σ) a $(\alpha)^* \in$ RV(Σ).*
- *RV(Σ) neobsahuje žádné jiné řetězce, do RV(Σ) tedy náleží pouze ty výrazy, které je možné vyrobit z $\emptyset$, ε a písmen abecedy Σ pravidly vymezenými výše.*

Definice 2 *Regulární výraz α je reprezentací jazyka $[\alpha]$, pro který platí:*

- $[\emptyset] = \emptyset$, *což je prázdný jazyk.*
- $[\varepsilon] = \{\varepsilon\}$, *což je jazyk obsahující pouze prázné slovo.*
- $[\alpha] = \{\alpha\}$ *pro všechny $\alpha \in \Sigma$, což jazyk zahrnující slovo α s délkou slova 1.*
- $[(\alpha + \beta)] = \{\alpha\} \cup \{\beta\}$, *což je sjednocení.*

- $[(\alpha \cdot \beta)] = \{\alpha\} \cdot \{\beta\}$, *což je zřetězení.*
- $[(\alpha)^*] = \{\alpha\}^*$, *což je iterace.*

Pro zjednodušení je možné v zápisu výrazu vynechávat některé přebytečné závorky — konkrétně ty vnější u operací zřetězení a sjednocení, vynecháváme i tečky u operace zřetězení. Také je možné vynechat vnější závorky u iterace jediného znaku.

Nejvyšší prioritu má operace $*$, druhou nejvyšší prioritu má operace $\cdot$ a nejnižší prioritu má operace $+$.

V této části práce je pro snadné porovnání a ilustraci používán tento regulární výraz, který je zároveň prvním příkladem, jak regulární výraz může vypadat:

Příklad 1
$(ab + b)^*ba$ ■

2.1.1 Reprezentace regulárního výrazu syntaktickým stromem

Regulární výraz jako takový je vlastně z určitého pohledu pouhým řetězcem znaků, který představuje předpis pro nalezení nějakého konkrétního řetězce. Aby bylo možné s regulárním výrazem provádět potřebné operace, je pouhý textový zápis znaků v řetězci nevhodný, protože tyto operace by bylo nutné provádět složitým způsobem, který by nutně zahrnoval opakované parsování informací z daného řetězce. Z tohoto důvodu je žádoucí regulární výraz ze všeho nejdříve převést na syntaktický strom. Operace parsování tak proběhne pouze jednou.

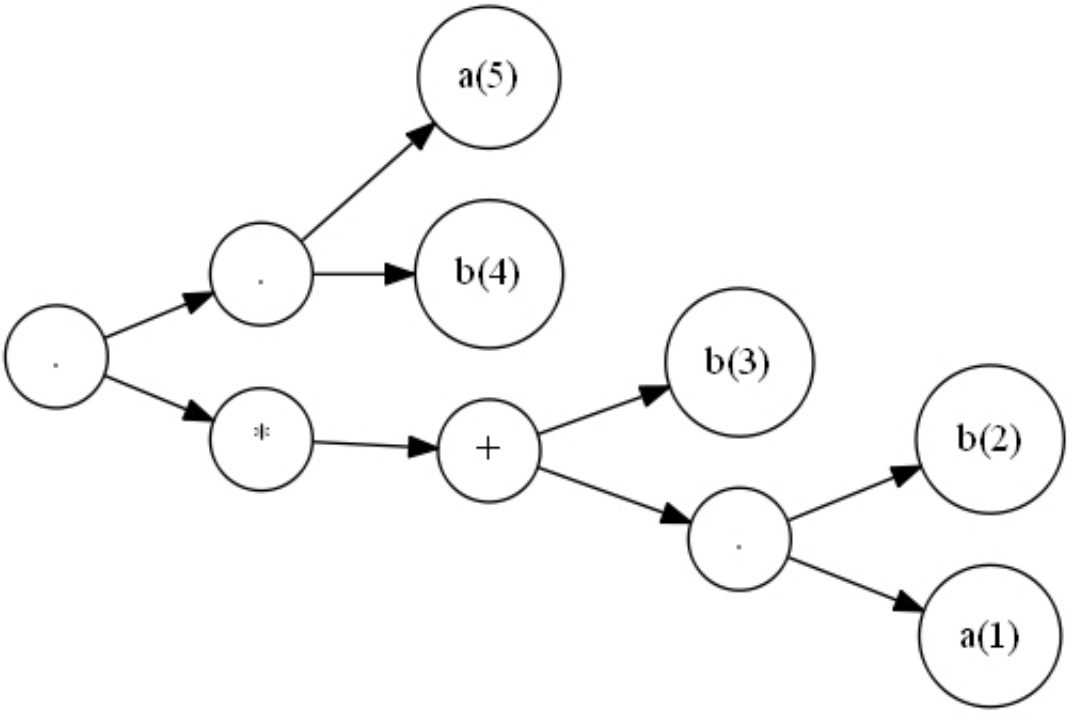
Definice 3 *Syntaktický strom je strom, kde:*

- *každý uzel je ohodnocen prvkem z* Σ*, přičemž vnitřní uzly jsou ohodnoceny operacemi* $\{+, \cdot, *\}$ *a listové uzly jsou ohodnoceny operandy* $\{\emptyset, \varepsilon, x\}$*,*
- *každý uzel má právě jednoho rodiče, kromě kořene, který rodiče mít nemůže,*
- *každý vnitřní uzel má minimálně 1 a maximálně 2 potomky,*
- *uzel, který nemá žadné potomky, je uzlem listovým,*
- *ve stromu existuje cesta z kořene pro všechny* $\alpha \in \Sigma$*, přičemž symboly abecedy se mohou opakovat, tudíž pro jeden symbol může existovat více cest.*

Pro převod regulárního výrazu v textové podobě do datové struktury syntaktický strom je využívána již výše uvedená priorita operací v kombinaci s uzávorkováním. Parsování tedy probíhá tak, že se jednoduše postupuje zleva znak po znaku, přičemž je do mezipaměti ukládána informace o tom, jaká operace je právě prováděna. Pokud je výraz v závorce, řeší se nejprve operace pro celý výraz v této závorce. Pak se rekurzivně vstoupí i do této závorky a celá operace se opakuje právě pro tento menší problém. Tento postup se nazývá rekurzivní sestup (recursive descent).

Pokud tedy například parser narazí na symbol reprezentující operand, který je následován operacemi $*$, $\cdot$ a speciálním znakem (, vytvoří vnitřní uzel typu $\cdot$, který bude mít jako levého potomka vnitřní uzel typu $*$ a jako pravého potomka kořen podvýrazu v následující závorce. Tito potomci pak budou mít další potomky, kteří jim budou přiděleni rekurzivně. Výsledný syntaktický strom vzniklý z ilustračního regulárního výrazu $(ab + b)^*ba$ je demonstrován na obrázku 1.

Poznámka 1 Jednotlivé znaky abecedy jsou ve výrazu ještě před samotným parsováním ohodnoceny pořadovým číslem. Je to specifikum potřebné pro tuto práci a jeho využití je objasněno v následujících kapitolách. V syntaktickém stromu se toto pořadí v listovém uzlu označuje číslem v závorce. Dalším specifikem pro turo práci je, že výsledný strom je v grafické podobě vykreslován zleva doprava, přičemž v publikacích je vykreslován spíše shora dolů.



Obrázek 1: Syntaktický strom vzniklý z regulárního výrazu $(ab + b)^*ba$

2.2 Konečný automat

Konečný automat je matematický výpočetní model. Je koncipován jako abstraktní stroj, který může být v jednom z konečného počtu stavů. Automat je v postupu zpracování vstupního slova vždy v jediném stavu, tedy v daném okamžiku se jedná o aktuální stav. Čtením slova se automat může změnit z jednoho stavu do druhého, a to pomocí přechodů. Konkrétní automat je vždy definován seznamem svých stavů a přechodovou funkcí pro každý stav a každý znak.

Výstupem automatu je informace, zda automat dané vstupní slovo přijal, či nikoliv. Konečný automat dokáže rozpoznávat pouze regulární jazyky. Bývá většinou zadán diagramem (grafem automatu) nebo tabulkou. Rozlišujeme základní dva druhy — deterministický a (zobecněný) nedeterministický.

2.2.1 Deterministický konečný automat

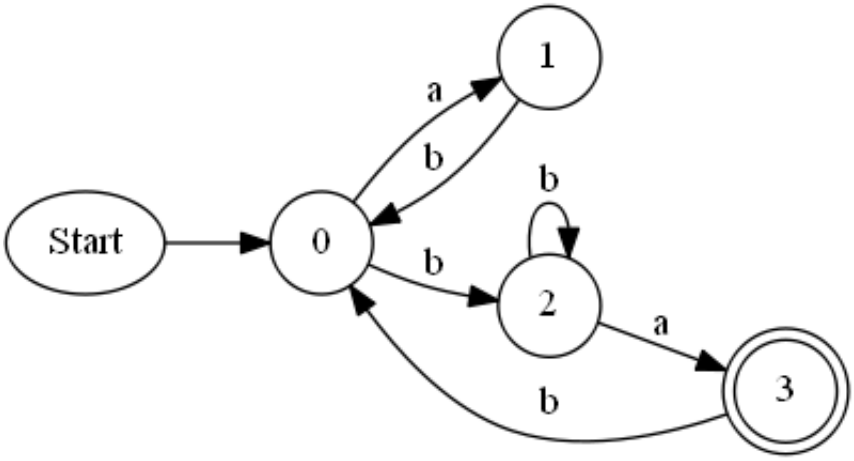
Deterministický konečný automat je specifický tím, že obsahuje právě jeden počáteční stav, koncových stavů však může mít více. Pro každý stav je jednoznačně určeno, který stav je následující pro daný znak přečtený ze slova na vstupu. Automat se přitom vždy nachází pouze v jednom ze stavů.

Definice 4 *Deterministický konečný automat je uspořádaná pětice* $A = (Q, \Sigma, \delta, q_0, F)$*, kde:*

- Q *je neprázdná konečná množina stavů,*
- Σ *je neprázdná konečná množina vstupní abecedy,*
- $\delta : Q \times \Sigma \to Q$ *je přechodová funkce,*
- $q_0 \in Q$ *je počáteční stav,*
- $F \subseteq Q$ *je neprázdná množina přijímajících stavů.*

Jazyk přijímaný konečným automatem je množinou všech slov, které automat přijímá, tedy těch slov, kterými automat dojde do některého z přijímajících stavů. Jazykem přijímaným automatem A tedy rozumíme jazyk L(A) = $\{w \in \Sigma^* \mid$ slovo w je přijímáno A$\}$.

Jazyk L $\subseteq \Sigma^*$ je regulární pokud jej lze rozpoznat konečným automatem nad abecedou Σ, tedy pokud existuje A takový, že L = L(A). DKA vzniklý z RV $(ab + b)^*ba$ je vyobrazen na obrázku 2.



Obrázek 2: Deterministický konečný automat přijímající slova jazyka $(ab + b)^*ba$

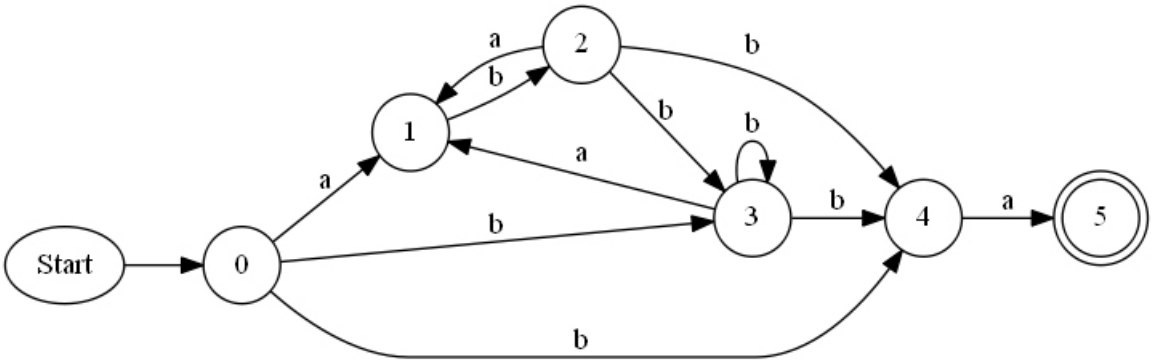
Poznámka 2 Jak je vidět na obrázku 2, v této práci používám označení počátečního stavu takové, že do něj vede šipka vycházející z kolečka s nápisem Start (nejedná se tedy o počáteční stav, tím je až stav, do kterého tato šipka vede). Dále si můžete povšimnout, že v grafu pro DKA není stav pro prázdný jazyk. Do tohoto stavu „pomyslně“ vedou přechody, které v grafu nejsou vyobrazeny.

2.2.2 Nedeterministický konečný automat

Nedeterministický konečný automat může narozdíl od deterministického mít více než jeden vstupní stav a také může přejít do více ůzných stavů prostřednictvím stejného symbolu. NKA vzniklý z RV $(ab + b)^*ba$ je vyobrazen na obrázku 3.

Definice 5 *Nedeterministický konečný automat je uspořádaná pětice $A = (Q, \Sigma, \delta, I, F)$, kde:*

- *Q je neprázdná konečná množina stavů,*
- *Σ je neprázdná konečná množina vstupní abecedy,*
- *$\delta : Q \times \Sigma \to P(Q)$ je nedeterministická přechodová funkce,*
- *$I \subseteq Q$ je množina počátečních stavů,*
- *$F \subseteq Q$ je neprázdná množina přijímajících stavů.*



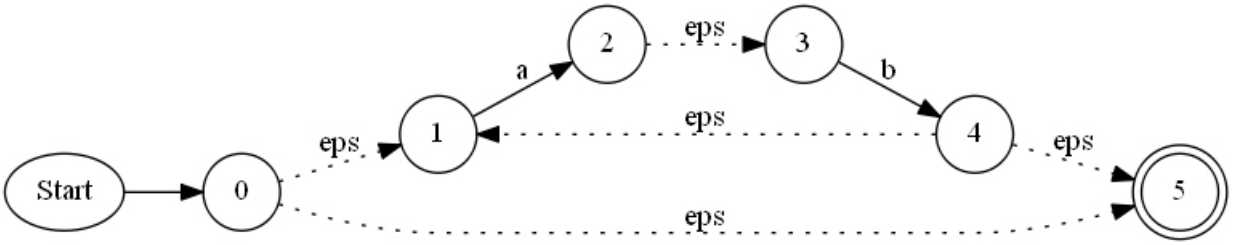
Obrázek 3: Nedeterministický konečný automat přijímající slova jazyka $(ab + b)^*ba$

2.2.3 Zobecněný nedeterministický konečný automat

Definice zobecněného nedeterministického konečného automatu je velice podobná, v přechodové funkci ale navíc figuruje symbol ε. Díky tomu může automat měnit svůj stav, aniž by zrovna četl symbol na vstupu. ZNKA vzniklý z RV $(ab + b)^*ba$ je vyobrazen na obrázku 4.

Definice 6 *Zobecněný nedeterministický konečný automat je uspořádaná pětice $A = (Q, \Sigma, \delta, I, F)$, kde:*

- *Q je neprázdná konečná množina stavů,*
- Σ *je neprázdná konečná množina vstupní abecedy,*
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$ *je nedeterministická přechodová funkce,*
- $I \subseteq Q$ *je množina počátečních stavů,*
- $F \subseteq Q$ *je neprázdná množina přijímajících stavů.*



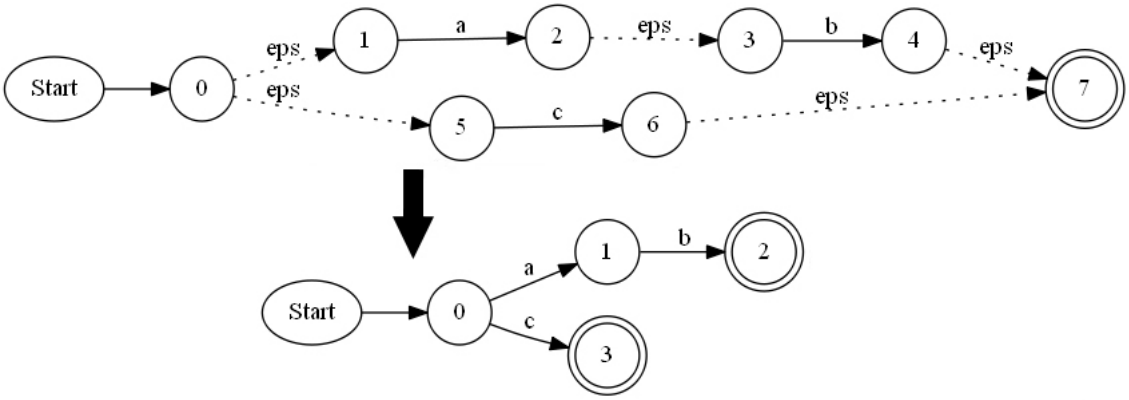
Obrázek 4: Zobecněný deterministický konečný automat přijímající slova jazyka $(ab)^*$

2.3 Převod (zobecněného) nedeterministického konečného automatu na deterministický konečný automat

V dalších částech této práce se setkáme s problémem, že některé algoritmy převádějí regulární výraz reprezenetovaný syntaktickým stromem do tvaru zobecněného nedeterministického konečného automatu, některé do nedeterministického konečného automatu a některé přímo do detereministického konečného automatu. Aby bylo takovéto převádění mezi sebou porovnatelné, potřebujeme výstup těchto algoritmů sjednotit. Jelikož jsme prostřednictvím některého z algoritmů schopni dosáhnout deterministického konečného automatu, bude konečný detereministický automat cíl, kterého chceme dosáhnout při aplikaci každého z těchto algoritmů. Převod ZNKA na DKA pro RV $(ab + c)$ je ilustrován na obrázku 5.

Potřebujeme tedy i algoritmus na převod (zobecněného) nedeterministického automatu na deterministický konečný automat. Toho dosáhneme standardní podmnožinovou konstrukcí:

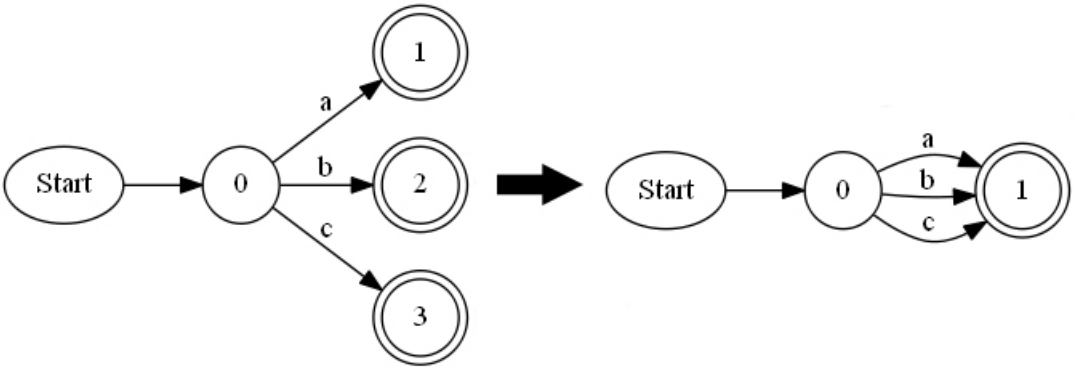
1. Sestrojíme stav, který reprezentue množinu I počátečních stavů nedeterministického automatu A.
2. Dokud máme v sestrojovaném automatu A' stavy s nadefinovanými přechody, zvolíme jeden takový stav q a znak x. Pro všechny stavy odpovídající q najdeme všechny možné přechody znakem x v A a shrneme je do nové množiny stavů q', přičemž tato množina se již v nově konstruovaném automatu může vyskytovat.
3. Pokud nový stav reprezentuje množinu, ze které se v A můžeme dostat $\varepsilon - p\check{r}echody$ do některého z koncových stavů F, označíme jej také jako koncový.



Obrázek 5: Příklad převodu zobecněného deterministického konečného automatu přijímající slova jazyka $(ab + c)$ na odpovídající deterministický konečný automat.

2.4 Minimalizace konečného atomatu

Konečný automat je minimální, pokud k němu neexistuje žádný ekvivalentní konečný automat, který by obsahoval méně stavů.



Obrázek 6: Příklad minimalizace deterministického konečného automatu přijímajícího slova jazyka $(a + b + c)$.

Pokud máme automat, který není minimální, musíme jej minimalizovat. Algoritmus minimalizace probíhá takto:

1. Máme automat $A = (Q, \Sigma, \delta, q_0, F)$.
2. Odstraníme všechny nedosažitelné stavy včetně přechodů vedoucích z nich.
3. Odstraníme všechny nadbytečné stavy $q \in Q$. Stav $q \in Q$ je nadbytečný, pokud neexistuje žádné slovo $w \in \Sigma^+$ a žádný stav $q_f \in F$ takový, že $\langle q, w \rangle \rightarrow^* \langle q_f, \varepsilon \rangle$. Zjednodušeně řečeno budeme postupně nalézat stavy, které vedou do koncových stavů a pak stavy, které vedou do stavů, které vedou do koncových stavů a tak pořád dokola. Tím získáme množinu všech stavů, které nejsou nadbytečné, díky čemuž zjistíme, které nadbytečné jsou.

Příklad minimalizace DKA přijímjícího slova jazyka $(a + b + c)$ si prohlédněte na obrázku 6.

2.5 Normovaný tvar konečného atomatu

Automat v normovaném tvaru je deterministický konečný automat bez nedosažitelných stavů, přičemž všechny stavy tohoto automatu jsou označeny jednoznačně.

Převodem konečných automatů do normovaného tvaru se zjednoduší jejich porovnávání, protože dva ekvivalentní normované automay jsou stejné i z hlediska označení stavů, odpadá tedy nutnost porovnávat různé kombinace uspořádaných dvojic stavů.

Algoritmus pro převod deterministického konečného automatu do normovaného tvaru:

1. Předpokládáme obecnou abecedu Σ s úplným uspořádáním prvků.
2. Na začátku jsou všechny stavy neoznačené a nezpracované.
3. Počáteční stav se označí číslem 1.
4. Dokud nejsou všechny označené stavy zpracované, opakuje se tato činnost:

   Vybere se označený nezpracovaný stav q s nejmenším číslem.

   Postupně se pro $j = 1, 2, \ldots, n$ opakuje tato aktivita: Jestliže stav q', pro který platí $\delta(q, a_j) = q'$, je neoznačený, potom se stav q' označí dosud nepoužitým nejmenším číslem.

   Stav q se prohlásí za zpracovaný.

Příklad výsledku převodu DKA do normovaného tvaru je na obrázku 7.

Obrázek 7: Příklad převodu deterministického konečného automatu přijímajícího slova jazyka $(a + b + c)$ do normovaného tvaru.

3 Algoritmy pro převod

Věta 1 *Ke každému jazyku, který je reprezentován regulárním výrazem, je možné sestavit konečný automat, který tento jazyk přijímá.*

V předchozích kapitolách bylo popsáno co je to regulární výraz a co je to konečný automat. V obou případech se pohybujeme ve třídě regulárních jazyků. Oba případy taktéž zajišťují základní funkci ověření, zda vybrané slovo patří do daného jazyka, či nikoliv. Z toho vyplývá, že regulární výrazy jsou převeditelné na konečné automaty a naopak a to při zachování generovaného/přijímaného jazyka.

V této práci je pozornost upřena k první variantě převodu, tedy převodu regulárního výrazu na konečný automat. Pro tento účel existují různé algoritmy. Některé z nich byly pro tuto práci vybrány, aby byly popsány, naimplementovány a porovnány, což je obsaženo v následujících kapitolách.

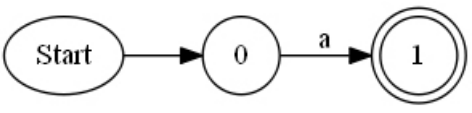
3.1 Thompsonův algoritmus

Algoritmus je založen na postupné dekompozici daného výrazu na menší části podle regulárních operací. Využíváme při tom faktu, že třída regulárních jazyků je uzavřena na operaci zřetězení, sjednocení a iteraci.

Věta 2 *Jestliže jazyky $L_1, L_2 \subseteq \Sigma$ jsou regulární, pak také jazyky $L_1 \cdot L_2, L_1 + L_2$ a L_1^* jsou regulární.*

Vezmeme tedy regulární výraz reprezentovaný syntaktickým stromem a tento strom začneme rekurzivně procházet. Při průchodu každým uzlem analogicky sestavujeme zobecněný nedeterministický automat a to v závislosti na tom, zda se jedná o uzel vnitřní, či listový a také na tom, jaký operand či operátor daný uzel obsahuje.

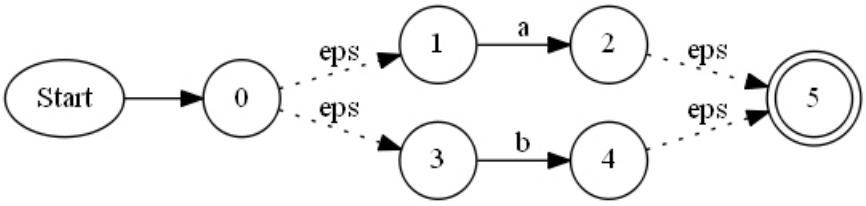
Nejjednodušším případem je listový uzel obsahující symbol. Pro takovýto případ jednoduše ve zobecněném nedeterministickém automatu vytvoříme dva stavy, přičemž z prvního stavu bude vést přechod do stavu druhého právě pro tento znak obsažený v listovém uzlu stromu (Obrázek 8).



Obrázek 8: Zobecněný deterministický konečný automat přijímající slova jazyka obsahujícího jediný operand (a).

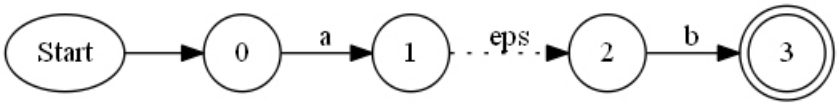
Druhým typem uzlu, na který můžeme při průchodu narazit, je sjednocení. V tomto případě v konečném automatu vytvoříme nový počáteční a koncový stav. Z počátečního stavu vedeme ε-přechody do počátečních stavů potomků, podobně vedeme i ε-přechody z koncových stavů

potomků do nového koncového stavu. Z počátečních a koncových stavů potomků se tedy stávají stavy „nepočáteční“ a „nekoncové“ (Obrázek 9).



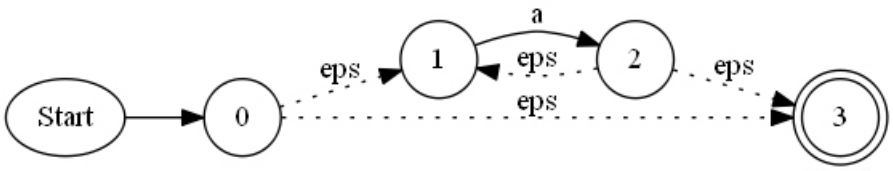
Obrázek 9: Zobecněný deterministický konečný automat přijímající slova jazyka $(a + b)$.

Třetím typem je zřetězení. V tomto případě v konečném automatu vedeme ε-přechod z koncového stavu levého potomka do počátečního stavu pravého potomka. Podobně jako v předchozím případě, z koncového stavu levého potomka se stává „nekoncový“ a z počátečního stavu pravého potomka se stává „nepočáteční“ (Obrázek 10).



Obrázek 10: Zobecněný deterministický konečný automat přijímající slova jazyka (ab).

Posledním druhem uzlu, na který můžeme v našem syntaktickém stromu narazit, je iterace. V konečném automatu vytvoříme nový počáteční a nový koncový stav. ε-přechody pak vedeme z nového počátečního stavu do nového koncového, z nového počátečního do původního počátečního, z původního koncového do nového koncového a také z původního koncového do původního počátečního. Z původního počátečního stavu se tedy stává „nepočáteční“ a z původního koncového stavu se stává „nekoncový“ (Obrázek 11).

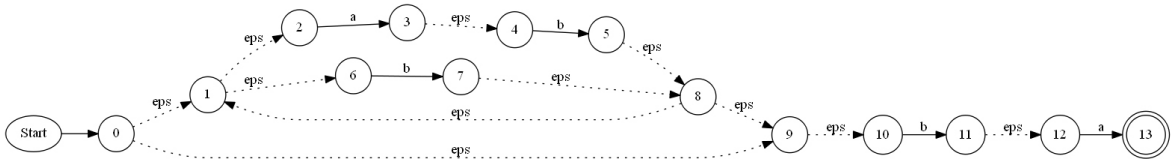


Obrázek 11: Zobecněný deterministický konečný automat přijímající slova jazyka $(a)^*$.

Postupnou aplikací těchto operací tak, že je budeme provádět od nejmenších problémů po největší, převedeme celý regulární výraz na konečný automat. Tento konečný automat bude přijímat stejný jazyk, jako generoval původní regulární výraz. Slovo w je pak přijímáno automatem A právě tehdy, když v grafu automatu existuje sled w začínající v počátečním stavu q_0 a končící v některém z koncových stavů.

Výstupem je zobecněný nedeterministický automat, který je pak možno převést na deterministický, ten minimalizovat a převést do normovaného tvaru.

Jak vypadá zobecněný nedeterministický konečný automat pro náš regulární výraz $(ab+b)^*ba$ je znázorněno na Obrázku 12.



Obrázek 12: Zobecněný nedeterministický konečný automat přijímající slova jazyka $(ab+b)^*ba$.

3.2 Berry/Sethi deriváty

Druhým algoritmem pro převod regulárního výrazu do deterministického konečného automatu je algoritmus pracujícími s deriváty, což jsou, zjednodušeně řečeno, operace levého kvocientu podle všech znaků abecedy daného výrazu aplikované nejprve na počáteční, a pak i na všechny nově vzniklé výrazy. Každý takto nově vzniklý výraz reprezentuje nový stav. Znak, podle kterého se prováděla operace levého kvocientu pak tvoří přechod z původního do nově vzniklého stavu. Pokud tedy konečný automat provádí přechod pod symbolem a, je na výraz E odpovídající stavu, ze kterého je přecházeno, provedena operace levého kvocientu $a\backslash E$. Výsledkem této operace je výraz E' odpovídající stavu, do kterého je pod daným symbolem přecházeno (tento stav je nově vytvořen). Pokud je výsledkem operace levý kvocient výraz, který již označuje nějaký existující stav, nový stav se již nevytváří, pouze se přidá přechod do tohoto již existujícího stavu.

Tento algoritmus tedy využívá funkci zajišťující operaci levého kvocientu, která podle syntaktického stromu výrazu E vytvoří nový syntaktický strom výrazu E', který je právě výsledkem operace levý kvocient podle vybraného znaku abecedy Σ.

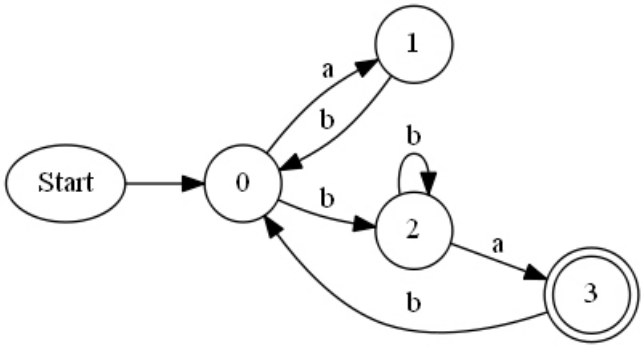
Ještě před samotným prováděním operace levého kvocientu potřebujeme znát funkci δ. Tato funkce slouží ke zjištění, zda výraz E obsahuje prázdné slovo, tedy ε. Pokud výraz E prázdné slovo ε obsahuje, je výsledkem operace $\delta(E)$ výraz E' obsahující ε. Řídíme se při tom těmito pravidly:

- $\delta(\emptyset) = \emptyset$
- $\delta(\varepsilon) = \varepsilon$
- $\delta(a) = \emptyset$
- $\delta(E + F) = \delta(E) + \delta(F)$
- $\delta(E \cdot F) = \delta(E) \cdot \delta(F)$
- $\delta(E^*) = \varepsilon$

Nyní celý strom rekurzivně projdeme a při návštěvě každého uzlu provedeme operaci levého kvocientu $a\backslash E$ podle těchto pravidel:

- $a\backslash\emptyset = \emptyset$
- $a\backslash a = \varepsilon, a\backslash b = \emptyset$ pro $b \neq a$
- $a\backslash(E + F) = a\backslash E + a\backslash F$
- $a\backslash(E \cdot F) = a\backslash E \cdot F + \delta(E) \cdot a\backslash F$
- $a\backslash(E^*) = a\backslash E \cdot E^*$

Automat je tedy sestaven díky vytvoření derivátů počátečního stavu, následně vytvořením derivátů těchto derivátů, a tak dále, dokud již nevznikají nové deriváty.



Obrázek 13: Deterministický konečný automat přijímající slova jazyka $(ab + b)^*ba$.

Po aplikaci tohoto algoritmu tedy získáme přímo deterministický konečný automat. Pro výraz $(ab + b)^*ba$ obdržíme automat o čtyřech stavech, který je vyobrazen na obrázku 13. Jeho jednotlivé stavy reprezentují tyto regulární výrazy:

- Stav 0 = $(ab + b)^*ba$ — počáteční stav obsahující původní výraz E.
- Stav 1 = $b(ab + b)^*ba$.
- Stav 2 = $(ab + b)^*ba + a$.
- Stav 3 = $b(ab + b)^*ba + \varepsilon$ — stav obsahující ε, tedy koncový.

Automat je nyní možné ještě minimalizovat a následně převést na normovaný tvar, ale v tomto případě již je rovnou minimalizovaný a i v normovaném tvaru.

3.3 Berry/Sethi odlišné znaky

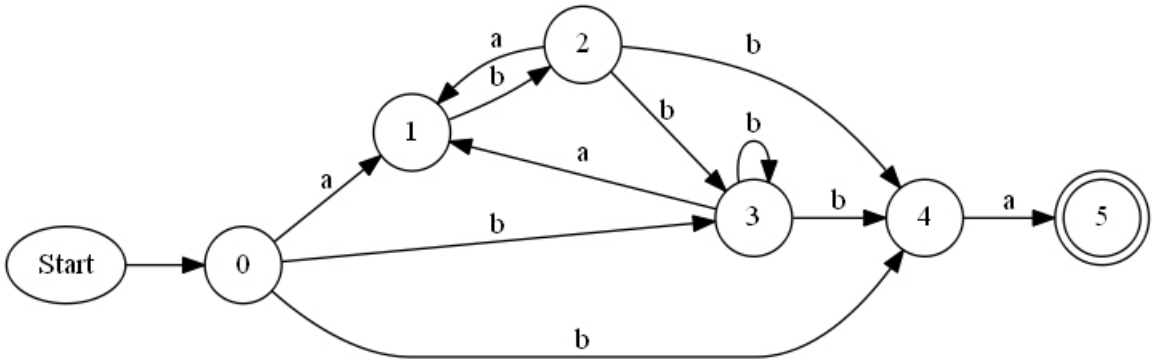
Další algoritmus těchto autorů i nadále využívá deriváty, ale odpadá zde výrazná zátěž v podobě porovnávání jazyka generovaného dvěma regulárními výrazy [1]. Hlavní odlišností je, že se zde pracuje s očíslovanými vstupními znaky abecedy. Regulární výraz $(ab + b)^*ba$ tedy přepíšeme na $(a_1b_2 + b_3)^*b_4a_5$. Zde jsme se tedy dostali k objasnění toho, proč je při parsování regulárního výrazu v textové podobě na syntaktický strom v listových uzlech za znak symbolu do závorky uváděno i pořadové číslo. Jedná se právě o toto očíslování.

Věta 3 *Převod pomocí odlišných znaků se provede postupem od regulárního výrazu E do DKA D takto:*

$$E \xrightarrow[očísluj\ symboly]{} E' \rightarrow M' \xrightarrow[odznač\ symboly]{} M \xrightarrow[konstrukce\ podmnožiny]{} D$$

Stavy automatu nyní vytvoříme z každého pořadím označeného symbolu. K nim přidáme jeden počáteční stav. Nyní nám zbývá vytvořit přechody. Postupně tedy procházíme jednotlivé stavy a na jejich odpovídající regulární výraz E postupně aplikujeme operaci levého kvocientu pro všechny označené znaky z abecedy Σ. Pokud je výsledkem této operace cokoliv jiného, než $\emptyset$, pak vedeme přechod z daného stavu do stavu odpovídajícímu znaku, podle kterého operace levého kvocientu byla prováděna. Nutno podotknout, že do daného stavu odpovídajícímu jednotlivému znaku z abecedy Σ vstupují pouze přechody pro právě tento znak. Například do stavu c_3 mohou vstupovat pouze přechody označené pouze symbolem c. Jako koncový je pak označen ten stav, který je následníkem jiného stavu a výsledkem aplikace operace δ na derivát E' předchozího stavu vytvořenou podle symbolu, kterým je označen následník, je ε.

Nyní již máme k dispozici jak stavy, tak přechody pro vytvoření nedeterministického konečného automatu. Pro výraz $(a_1b_2 + b_3)^*b_4a_5$ tedy dostaneme nedeterministický konečný automat se šesti stavy, který je vyobrazen na obrázku 14.



Obrázek 14: Nedeterministický konečný automat přijímající slova jazyka $(a_1b_2 + b_3)^*b_4a_5$.

Jelikož se jedná o nedeterminisitický konečný automat, je nutné jej pak běžným způsobem převést na deterministický konečný automat, případně jej pak dále minimalizovat a převést na normovaný tvar.

3.4 Berry/Sethi rychlý algoritmus

Zde je narozdíl od předchozích dvou algoritmů zcela upuštěno od náročného počítání levého kvocientu. Zato se zde objevují dvě novinky, kterými jsou množina *first* a množina *follow*.

Věta 4 *First(E) je množina všech znaků, které se mohou vyskytovat jako první písmeno všech slov generovaných L(E)* $(\{a \in E\}\ \exists w : aw \in L(E))$.

Množinu *first* tedy přiřadíme každému vrcholu v syntaktickém stromu. Podobně jako v předchozích alogoritmech je k tomu potřeba rekurzivně projít celý strom od kořene k listům a pro každý uzel tuto množinu zaznamenat. Pro to, co má a nemá být v množině *first*, jsou opět dána pravidla.

Pro *first(E)*, platí:

- $first(\emptyset) = \emptyset$
- $first(\varepsilon) = \emptyset$
- $first(a) = \{a\}$
- $first(E + F) = first(E) + first(F)$
- $first(E \cdot F) = \begin{cases} first(E) & \text{když } \varepsilon \notin L(E), \\ first(E) \cup first(F) & \text{když } \varepsilon \in L(E). \end{cases}$
- $first(E^*) = first(E)$

Podobným způsobem musíme vytvořit množinu *follow*.

Věta 5 *Follow(E) je množina všech znaků, které se mohou objevit za daným symbolem, pokud slovo tento symbol obsahuje* $(\{a \in E\}\ \exists w : wa \in L(E))$.

Pokud tedy již procházíme celý strom, abychom pro každý uzel uložili množinu *first*, rovnou při návštěvě daného uzlu určíme a uložíme i množinu *follow*. K tomu opět máme k dispozici definici pravidel:

Definice 7 *Mějme regulární výraz E. Pro operaci follow pro výraz E s parametrem S, značeno* $follow(E, S)$*, platí:*

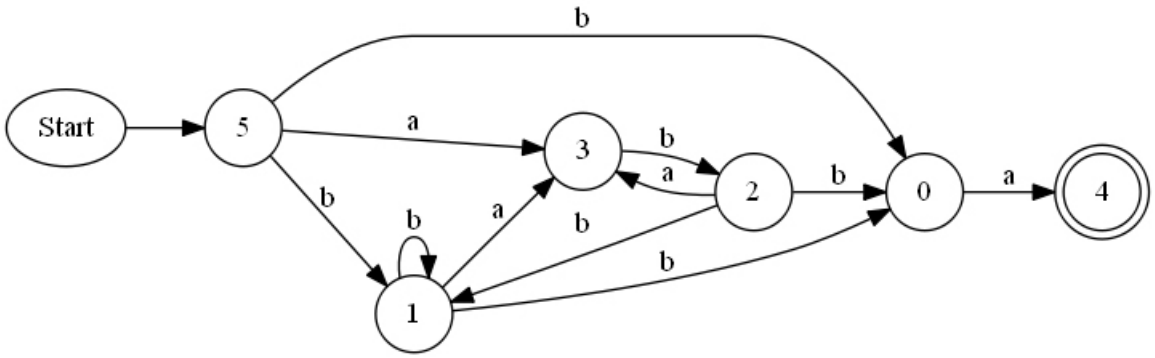
- $follow(\emptyset, S) = \emptyset$
- $follow(\varepsilon, S) = \emptyset$
- $follow(a, S) = \{S\}$
- $follow(E + F, S) = follow(E, S) + follow(F, S)$
- $follow(E \cdot F, S) = follow(E, S) \cup follow(E, first(F) + \delta(F) \cdot S)$
- $follow(E^*, S) = follow(E, s\ \cup first(E))$

V tomto bodě je ještě potřeba ošetřit jeden speciální případ. V tomto algoritmu budeme pracovat s tzv. speciálním **koncovým znakem**. Tento koncový znak je označen jako !. Je to obdoba <END OF FILE>. Nesmí se vyskytovat nikde jinde než na konci výrazu a to pouze jednou. Tento koncový znak do výrazu přidáme ještě před začátkem vykonávání algoritmu.

V konečném důsledku tedy budeme celý algoritmus provádět pro výraz $E!$. Tento speciání znak nám poslouží pro určení koncových stavů.

Po jednom průchodu a vygenerování množin *first* a *follow* již máme vše potřebné k sestavení nedeterministického konečného automatu. Stavy automatu zde budou reprezentovány všemi listovými uzly syntaktického stromu. K nim přidáme jeden nový počáteční stav. Přechody z počátečního stavu získáme z množiny *first(E!)*. Pokud by se v této množině objevil speciální koncový znak !, je počáteční stav zároveň koncovým. Všechny další přechody získáme z množiny *follow* každého listového uzlu. Pokud by se v této množině objevil speciální znak !, jedná se o koncový stav.

Pro výraz $(ab+b)^*ba$ obdržíme nedeterministický konečný automat se šesti stavy vyobrazený na ilustraci 15. Není náhoda, že se jedná o identický nedeterministický konečný automat, jaký jsme dostali u předchozího algoritmu (po převodu do normovaného tvaru by měl i naprosto stejný graf).



Obrázek 15: Nedeterministický konečný automat přijímající slova jazyka $(a_1b_2 + b_3)^*b_4a_5$.

3.5 Brüggemann-Klein algoritmus

V tomto algoritmu je operace *follow* oproti předchozímu algoritmu mírně modifikována a přibývá i operace *last*.

Funkce *follow(E)* mapuje pozice E na podmnožiny pozicí E:

- $follow(\emptyset, S)$ — E nemá pozice
- $follow(\varepsilon, S)$ — E nemá pozice
- $follow(a, S) = \emptyset$
- $follow(E + F, S) = \begin{cases} follow(E, S) & \text{když } S \in pos(E), \\ follow(F, S) & \text{když } S \in pos(F) \end{cases}$

- $follow(E \cdot F, S) = \begin{cases} follow(E,S) & \text{když } S \in pos(E) - last(E), \\ follow(E,S) \cup first(F) & \text{když } S \in last(E), \\ follow(F,S) & \text{když } S \in pos(F) \end{cases}$
- $follow(E^*, S) = \begin{cases} follow(E,S) & \text{když } S \in pos(E) - last(E), \\ follow(E,S) \cup first(E) & \text{když } S \in last(E) \end{cases}$

Operace *last* je velmi podobná operaci *first*, která již byla definována výše u jednoho z předchozích alogoritmů. Řídí se těmito pravidly:

- $last(\emptyset) = \emptyset$
- $last(\varepsilon) = \emptyset$
- $last(a) = \{a\}$
- $last(E + F) = last(E) + last(F)$
- $last(E \cdot F) = \begin{cases} last(F) & \text{když } \varepsilon \notin L(F), \\ last(E) \cup last(F) & \text{když } \varepsilon \in L(F). \end{cases}$
- $last(E^*) = last(E)$

Pozice v automatu odpovídají symbolům ve výrazu. Vytvoříme podle nich stavy a taktéž přidáme nový počáteční stav. Přechody vytvoříme tak, že pro každé $a \in \Sigma$ zjistíme množinu *follow(E, a)*. Přechody z počátečního stavu se nacházejí v množině získané z *first(E)*. Všechny koncové stavy jsou v množině *last(E)*.

Tím získáme všechna potřebná data pro sestavení NKA. Pro výraz $(ab + b)^*ba$ obdržíme totožný NKA jako u předchozího algoritmu, tedy ten, který je zobrazen na obrázku 15.

3.6 Optimalizovaný Brüggemann-Klein algoritmus s nullable proměnnou

Když konvertujeme regulární výraz na syntaktický strom, tak, jak je již definováno výše, listové uzly tvoří znaky $\emptyset, \varepsilon$ a smboly abecedy, vnitřní uzly pak tvoří operátory $+$, $\cdot$ a $*$. Každý uzel v syntaktického stromu tedy odpovídá podvýrazu E_v ve ztahu k E.

V každém uzlu si pak připravujeme množiny *first*, *last* a v tomto případě nově také boolean proměnnou *nullable*. Ta nám určuje, zda podvýraz E_v ve vztahu k v obsahuje prázdné slovo ε. Dále zde máme množinu *follow*, kterou ale v tomto případě neukládáme do vrcholů stromu, ale máme pro to vytvořenu globální množinu, která pro každé $x \in pos(E)$ obsahuje podmnožinu *follow(x)*. Na strom tedy aplikujeme postorder průchod a na základě informací dostupných z účasti v každém uzlu, ve kterém se zrovna nacházíme, shormažďujeme potřebné informace. Při tom se řídíme těmito pravidly:

- v je listový uzel označený $\emptyset$:

  $nullable(v) := false$;

  $first(v) := \emptyset$;

  $last(v) := \emptyset$;

- v je listový uzel označený ε:

  $nullable(v) := true$;

  $first(v) := \emptyset$;

  $last(v) := \emptyset$;

- v je listový uzel označený x:

  $nullable(v) := false$;

  $follow(x) := \emptyset$;

  $first(v) := \{x\}$;

  $last(v) := \{x\}$;

- v je vnitřní uzel označený $+$:

  $nullable(v) := nullable(levyPotomek)$ or $nullable(pravyPotomek)$;

  $first(v) := first(levyPotomek) \cup first(pravyPotomek)$;

  $last(v) := last(levyPotomek) \cup last(pravyPotomek)$;

- v je vnitřní uzel označený $\cdot$:

  $nullable(v) := nullable(levyPotomek)$ and $nullable(pravyPotomek)$;

  pro každé $x \in$ last(levyPotomek) proveď:

  follow(x) := $follow(x) \cup first(pravyPotomek)$;

  $$\text{first(v)} := \begin{cases} first(levyPotomek) \cup first(pravyPotomek) & \text{když } nullable(levyPotomek), \\ first(levyPotomek) & \text{když } !nullable(levyPotomek). \end{cases}$$

  $$\text{last(v)} := \begin{cases} last(levyPotomek) \cup last(pravyPotomek) & \text{když } nullable(pravyPotomek), \\ last(pravyPotomek) & \text{když } !nullable(pravyPotomek). \end{cases}$$

- v je vnitřní uzel označený $*$:

  $nullable(v) := true$;

  pro každé $x \in$ last(potomek) proveď:

  follow(x) := $follow(x) \cup first(potomek)$;

  $first(v) := first(potomek)$;

  $last(v) := last(potomek)$;

Všechny potřebné množiny k sestavení konečného automatu máme po průchodu stromem nachystány. Sestavení automatu pak probíhá identicky, jako v algoritmu v předchozí kapitole. Díky zavedení *nullable* proměnné a globální množině pro ukládání podmnožin vygenerovaných v jednotlivých vrcholech, jsou všechny námi žádané množiny připraveny již při prvním průchodu.

3.7 Optimalizovaný Bruggemann-Klein algoritmus s nullable proměnnou a vstupem ve star normal form

Všechna sjednocení v předchozím algoritmu označená jako červená a zelená jsou sjednocením disjunktních množin, což je možné udělat v konstantním čase (to proto, že $pos(E) \cap pos(F) = \emptyset$ když $(E + F)$ nebo $(E \cdot F)$ jsou podvýrazy E). [2]

Pouze sjednocení množin označených modře nemusí být vždy sjednocením množin disjuktních. Ohvězdičkovaný podvýraz H^* výrazu E přidá prvky z *first(H)* do *follow(H, x)* pro $x \in$ *last(H)*, ale některé prvky z *first(H)* se v množině *follow(H, x)* již mohou nacházet pro některá $x \in$ *last(H)*, jako je tomu například u výrazu $(a^*b^*)^*$.

Musíme tedy zajistit, aby vstupem byl vždy takový výraz, že i sjednocení označené modře bude sjednocením disjunktních množin. Výraz, který toto splňuje, je v **SNF** (star normal form). Pokud budeme schopni zajistit, aby vstupem byl vždy výraz v SNF, pak jsme také schopni zajistit, že náš algoritmus bude pracovat s časovou složitostí $O(size(M_E))$. Vzhledem k tomu, že na vstupu můžeme obdržet jakýkoliv regulární výraz, který ve star normal form není, je nutné zajistit, aby na vstup byla vždy nejprve aplikována transformace právě do SNF.

Definice 8 *Regulární výraz E je ve star normal form pokud pro každý ohvězdičkovaný podvýraz H^* výrazu E platí follow(H, last(H)) $\cap first(H) = \emptyset$ a $\varepsilon \notin L(H)$.*

3.7.1 Transformace regulárního výrazu do star normal form

Věta 6 *Pro každý regulární výraz E existuje regulární výraz $E^\bullet$ takový, že:*

1. $M_E = M_{E^\bullet}$,
2. $E^\bullet$ *je ve star normal form,*
3. $E^\bullet$ *může být z E vytvořeno v lineárním čase.*

Jako mezikrok popíšeme převod výrazu E^* na výraz $E^{\circ *}$ se stejným konečným automatem tak, že podmínky uvedené v Definici 8 jsou splněny alespoň na nejkrajnější úrovni, a to pro E°.

Pokud odstraníme z M_E všechny „zpětnovazební“ přechody vedoucí z koncových stavů (například z q_1) do stavů, do kterých je q_1 přímo napojeno, a pokud z q_1 uděláme nekoncový stav, pak je výstupní nedeterministický konečný automat pro výraz E° s $follow(E^\circ, last(E^\circ)) \cap first(E^\circ) = \emptyset$. Všechny „zpětnovazební“ přechody odstraněné při převodu M_E na M_{E° jsou znovu zavedeny v $M_{E^{\circ *}}$, tedy máme $M_{E^{\circ *}} = M_{E^*}$.

Definice 9 *Mějme regulární výraz E. Pro transformaci na E° platí:*

- $\emptyset^\circ = \emptyset$
- $\varepsilon^\circ = \emptyset$
- $a^\circ = \{a\}$
- $(E + F)^\circ = E^\circ + F^\circ$
- $(E \cdot F)^\circ = \begin{cases} E \cdot F & \text{když } \varepsilon \notin L(E) \text{ a zároveň } \varepsilon \notin L(F), \\ E^\circ \cdot F & \text{když } \varepsilon \notin L(E) \text{ a zároveň } \varepsilon \in L(F), \\ E \cdot F^\circ & \text{když } \varepsilon \in L(E) \text{ a zároveň } \varepsilon \notin L(F), \\ E^\circ + F^\circ & \text{když } \varepsilon \in L(E) \text{ a zároveň } \varepsilon \in L(F). \end{cases}$
- $E^* = E^\circ$

Substituce výrazu H^* výrazem $H^{\circ *}$ ponechává automat H^* neporušený, přičemž $follow(H^\circ, last(H^\circ)) \cap first(H^\circ) = \emptyset$.

Pokud tedy v E nahradíme každý ohvězdičkovaný podvýraz H^* za $H^{\circ *}$, obdržíme výraz $E^\bullet$ ve star normal form, přičemž $M_E = M_{E^\bullet}$.

Definice 10 *Mějme regulární výraz E. Pro transformaci na $E^\bullet$ platí:*

- $\emptyset^\bullet = \emptyset$
- $\varepsilon^\bullet = \varepsilon$
- $a^\bullet = \{a\}$
- $(E + F)^\bullet = E^\bullet + F^\bullet$
- $(E \cdot F)^\bullet = E^\bullet \cdot F^\bullet$
- $E^* = E^{\bullet \circ *}$

Zbývá nám zajistit, aby $E^\bullet$ bylo z E vypočítáno v lineárním čase. Pokud víme, že $E^\bullet$ je vytvořeno z $H^\bullet$ a $H^{\bullet\circ}$ pro podvýrazy H výrazu E, pak také víme, že můžeme $H^\bullet$ a $H^{\bullet\circ}$ počítat zároveň. K tomu slouží operace s těmito pravidly:

Definice 11 *Mějme regulární výraz E. Pro transformaci na $E^{\bullet\circ}$ platí:*

- $\emptyset^{\bullet\circ} = \emptyset$
- $\varepsilon^{\bullet\circ} = \emptyset$
- $a^{\bullet\circ} = \{a\}$

- $(E + F)^{\bullet\circ} = E^{\bullet\circ} + F^{\bullet\circ}$
- $(E \cdot F)^{\bullet\circ} = \begin{cases} E^{\bullet} \cdot F^{\bullet} & \text{když } \varepsilon \notin L(E) \text{ a zároveň } \varepsilon \notin L(F), \\ E^{\bullet\circ} \cdot F^{\bullet} & \text{když } \varepsilon \notin L(E) \text{ a zároveň } \varepsilon \in L(F), \\ E^{\bullet} \cdot F^{\bullet\circ} & \text{když } \varepsilon \in L(E) \text{ a zároveň } \varepsilon \notin L(F), \\ E^{\bullet\circ} + F^{\bullet\circ} & \text{když } \varepsilon \in L(E) \text{ a zároveň } \varepsilon \in L(F). \end{cases}$
- $E^* = E^{\bullet\circ}$

Aplikací této operace na regulární výraz E tedy obdržíme regulární výraz $E^{\bullet}$ ve star normal form takový, že $L(E) = L(E^{\bullet})$. U již zmíněného výrazu $(a^*b^*)^*$ by tedy transformace proběhla takto:

Příklad 2

1. $(a^*b^*)^{*\bullet} = (a^*b^*)^{\bullet\circ *}$
2. $(a^*b^*)^{\bullet\circ *} = (a^{*\bullet\circ} + b^{*\bullet\circ})^*$
3. $(a^{*\bullet\circ} + b^{*\bullet\circ})^* = (a^{\bullet\circ} + b^{\bullet\circ})^*$
4. $\underline{\underline{(a^{\bullet\circ} + b^{\bullet\circ})^* = (a + b)^*}}$

■

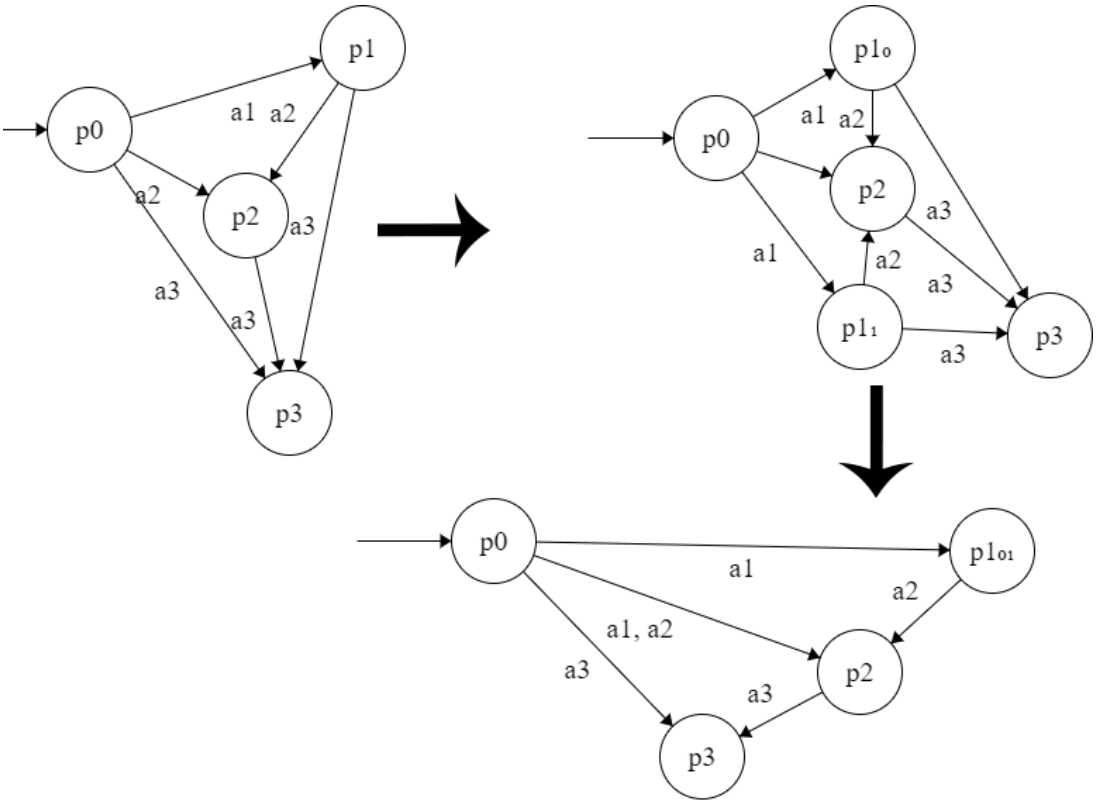
Výsledný výraz $E^{\bullet}$ ve star normal form pak dáme na vstup zrychlené verze algoritmu z předchozí kapitoly (zrychlenou verzí je zde myšlen totožný algoritmus se slučováním disjunktních množin *first*, *last* a *follow* v konstantním čase).

3.8 CFS algoritmus

V článku Translating regular expressions into small ε-free nondeterministic finite automata [4] je popsán CFS (common follow sets) algoritmus, který pracuje s dekompozicí výstupních přechodů stavů automatu. Každý stav automatu je zde nahrazen několika kopiemi, tyto kopie mají stejné vstupní stavy jako stav, ze kterého byly zkopírovány, ale výstupní přechody jsou určitým způsobem rozdistribuovány mezi tyto nově vzniklé kopie. V takovém automatu pak zjistíme, které kopie jsou totožné, a ty sloučíme. Při prvotním vytváření kopií stavů a distribucí jejich výstupních přechodů se počet přechodů zvyšuje, při následném slučování totožných stavů se počet přechodů snižuje. Může se stát, že díky této operaci dojde k tomu, že po sloučení totožných stavů dojde ke snížení celkového počtu přechodů.

Základem tohoto algoritmu je opět rekurzivní sestup. Při něm ale tentokrát provádíme dekompozici množiny *follow*, tzn. rozdělujeme následníky konkrétního stavu do několika podskupin. Tyto podskupiny jsou pak novým stavem v automatu, přičemž každá množina C přebírá přechody původního stavu s prvky C. Pokud je automat ve stavu x (pozice E), bude nový automat v jedné z vybraných podmnožin *follow(E, x)*, neboli v C. Každý přechod z x do každého $x' \in C$ je nahrazen přechody z C do každého C' patřícího do rozkládaného *follow(E, x)*.

Vzhledem k tomu můžeme po několika rozkladech obdržet stejnou C množinu (myšleno rozklady různých *follow* množin), což potenciálně vede ke snížení celkového počtu přechodů (viz Obrázek 16).



Obrázek 16: Počet přechodů se kvůli kopii původního stavu p1 nejprve zvýšil, ale díky sloučení stejných stavů se v konečném důsledku snížil.

Definice 12 *Nechť E je regulární výraz, daný svou sadou pozic pos(E). Systém CFS pro E je dán dekompozicí $dec(x) \subseteq pos(E)$ pro každé $x \in pos(E)$ a je nutné aby bylo splněno $dec(x) \neq \oslash$ a*

$$follow(E, x) = \bigcup_{C \in dec(x)} C$$

pro každé $x \in pos(E)$.

Stavy automatu tvoříme tak, že pro každou dekompozicí vzniklou množinu C vytvoříme v automatu dva stavy - jeden přijímající a druhý nepřijímající. K tomu ještě vytvoříme jeden počáteční stav, který bude odpovídat množině *first(E)* (ten bude (ne)přijímající podle toho, zda $\varepsilon \in L(E)$).

Pro každou pozici $x \in pos(C)$, kde C je odpovídající stavu, ze kterého budeme vést přechody, vedeme přechody do stavů odpovídajících jiným množinám C', a to takovým, které vznikly

dekompozicí *follow(E, x)*. Znak přechodu odpovídá pozici, pro kterou převod provádíme. Jelikož je ale pro každou množinu C' vytvořen přijímající i nepřijímající stav, vedeme stav do přijímající varianty stavu odpovídající množině C' pouze tehdy, je-li pozice obsažena v *last(E)*, v opačném případě je přechod veden do nepřijímající varianty tohoto stavu.

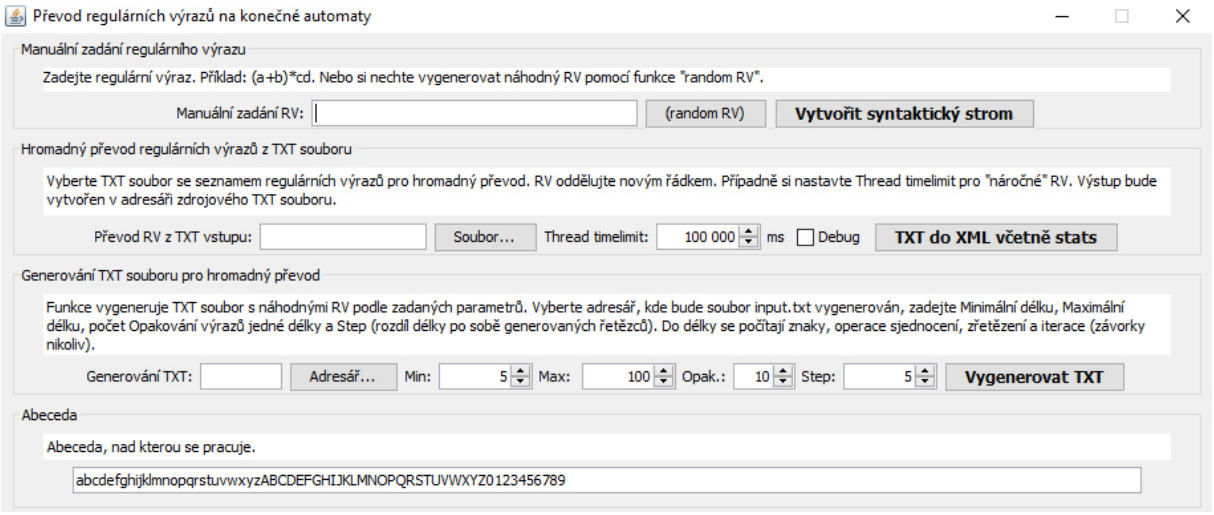
Jak konkrétně je ale dekompozice prováděna? Celý strom je nejprve rozdělen na podstromy t vzniklé z t_E. Toho je dosaženo tak, že stromy jsou opakovaně rekurzivně rozdělovány na dvě poloviny t_1 a t_2 tak, že každá polovina je velká alespoň z 1/3 původního stromu. Jakmile je celý strom rozdělený na podstromy, vracíme se v průchodu od listů směrem ke kořeni. Při každé návštěvě uzlu, kde při sestupu došlo k rozdělení stromu, se zpracuje *dec* množina pro pozice obsažené v podstromech t_1 a t_2. Pro podstrom t_1 se provádí jiné operace, než pro podstrom t_2. Jejich konkrétní podobu najdete v Kapitole 4.2.1.9, kde je popsána implementace tohoto algoritmu.

4 Implementace vybraných metod převodu regulárních výrazů na konečné automaty

Realizace implementace výše uvedených algoritmů probíhala v jazyce Java, verze SE 8. Aplikace zahrnuje jednoduché uživatelské rozhraní s možností zadávat vlastní regulární výrazy, nechat si regulární výrazy vygenerovat a hlavně je zde možnost vybrat vstupní TXT soubor obsahující regulární výrazy pro hromadný převod. Po manuálním převedení jednoho regulárního výrazu na syntaktický strom dojde k zobrazení grafické reprezentace tohoto stromu, stejně tak je grafickým výstupem reprezentován i jakýkoliv automat vzniklý aplikací jednotlivých algoritmů pro převod regulárního výrazu na konečný automat. U hromadného převodu není zobrazován grafický výstup, nýbrž jsou výsledné automaty ukládány do **XML**, díky čemuž je pak možné zjistit, zda jsou výsledné automaty stejné. Při použití hromadného převodu je také vygenerován soubor **stats.csv**, který obsahuje časové statistiky zaznamenané při vykonávání jednotlivých algoritmů.

4.1 Aplikace z pohledu uživatele

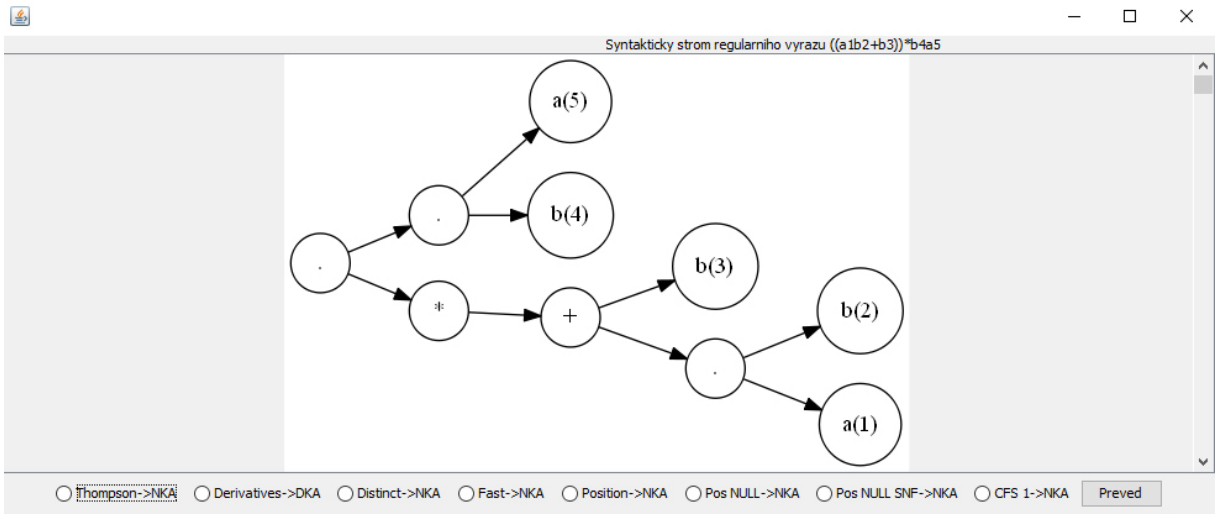
Po spuštění aplikace má uživatel k dispozici panel rozdělený podle funkčnosti na tři části (Obrázek 17).



Obrázek 17: Grafické uživatelské rozhraní aplikace.

4.1.1 Manuální zadání výrazu

V první části může uživatel zadat svůj vlastní regulární výraz a převést ho na syntaktický strom. Je zde také možnost nechat si vygenerovat náhodný výraz. Jakmile je výraz převeden na syntaktický strom, je zobrazena grafická reprezentace tohoto stromu v novém okně. V tomto novém okně lze dále pokračovat výběrem algoritmu prostřednictvím kterého bude uskutečněn převod do (Z)NKA/DKA (Obrázek 18).



Obrázek 18: Nabídka volby algoritmu, podle kterého se daný syntaktický strom převede na konečný automat. U každého algoritmu je vždy poznamenáno do jakého druhu konečného automatu bude syntaktický strom převeden.

4.1.2 Hromadný převod výrazů

V prostřední části okna aplikace je pole pro výběr TXT souboru z disku. Jedná se o vstupní TXT soubor obsahující regulární výrazy pro hromadný převod prostřednictvím všech dostupných (naimplementovaných algoritmů). Regulární výrazy jsou zde odděleny novým řádkem. Po vybrání souboru, nastavení časového limitu a kliknutí na spouštěcí tlačítko dojde k otevření nového okna, kde bude zobrazen průběh hromadného převodu. Jakmile hromadný převod úspěšně skončí, zobrazí se příslušná informace o dokončení. V adresáři, odkud pochází vstupní TXT soubor, nyní máme nový adresář obsahující XML reprezentace jednotlivých automatů a také soubor stats.csv obsahující naměřené hodnoty při hromadném převodu jednotlivými algoritmy.

4.1.2.1 Výstupní XML soubor při hromadném převodu

Každý regulární výraz zadaný v TXT souboru je převeden do minimalizovaného DKA. Po každém provedení konkrétního algoritmu jsou uloženy všechny výsledky v mezikrocích, tedy i předcházející NKA. XML soubor reprezentující automat vždy obsahuje všechny stavy. V těchto stavech jsou uloženy informace o čísle stavu, zda je počáteční či koncový a také jsou zde zaznamenány všechny přechody do jiných stavů. Příklad konkrétní XML reprezentace automatu je ve Výpisu 1.

```
<DKA>
   <stav>
      <cislo>0</cislo>
      <pocatecni>t</pocatecni>
      <koncovy>f</koncovy>
      <prechody>
```

```
        <prechod>
          <znak>a</znak>
          <kam>0</kam>
        </prechod>
        <prechod>
          <znak>b</znak>
          <kam>0</kam>
        </prechod>
      </prechody>
   </stav>
</DKA>
```

Výpis 1: Struktura XML výsledného minimalizovanéhoDKA.

4.1.2.2 Výstupní CSV soubor při hromadném převodu Soubor stats.csv obsahuje naměřené časy v milisekundách potřebné pro jednotlivé operace prováděné při vykonávání jednotlivých algoritmů. To znamená, že pro konkrétní výraz, který byl zpracován konkrétním algoritmem, známe čas potřebný pro převod z RV do NKA, z NKA do DKA, jak dlouho trvala minimalizace a pak také sumu těchto hodnot. Jsou zde zaznamenávány i další méně důležité statistiky potřebné pro testování při implementaci, jako například počet terminálních symbolů apod. Tyto hodnoty pak slouží k dalšímu zpracování, nejčastěji v MS Excel. Příklad souboru stats.csv je uveden na Obrázku 19.

	A	B	C	D	E	F	G	H	I	J
1	C. RV	term. znal	real delka	vel. abec	Alg	RV->SNF	RV->NKA	NKA->DKA	Mini	celkem
2	1	40	100	29	Thompsor	-	3,672996	16,23516	0,351822	20,25998
3	1	40	100	29	B/S Deriva	-	-	-	-	-
4	1	40	100	29	B/S Distin	-	-	-	-	-
5	1	40	100	29	B/S Fast	-	1,662228	0,657256	0,368654	2,688138
6	1	40	100	29	BK positio	-	4,344209	0,702003	0,206496	5,252708
7	1	40	100	29	BK positio	-	1,229942	0,541897	0,184328	1,956167
8	1	40	100	29	BK positio	0,094832	1,400723	0,678192	0,208549	2,287464
9	1	40	100	29	CFS1	-	2,307168	0,361265	0,162159	2,830592

Obrázek 19: Ukázka obsahu souboru stats.csv.

4.1.3 Generátor TXT souboru pro hromadný převod

Třetí část hlavního okna aplikace obsahuje funkci pro generování TXT souboru, který pak slouží jako vstup pro hromadný převod. Prvním krokem je volba adresáře kam bude TXT soubor vygenerován (jedná se o soubor, který bude pojmenován input.txt). Druhým krokem je nastavení

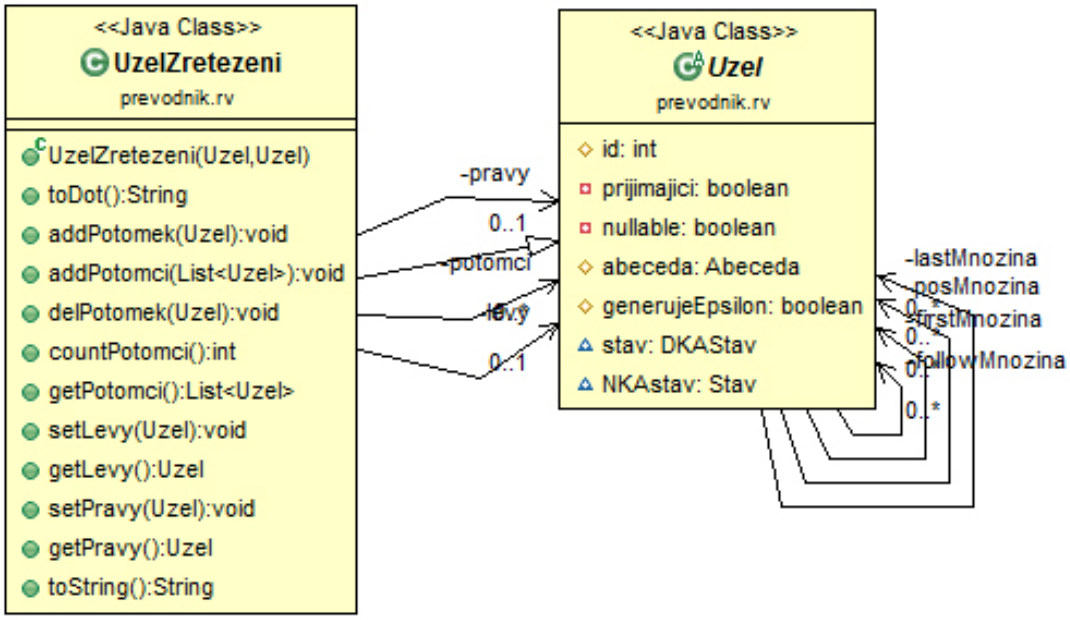
parametrů, které bude výsledný soubor splňovat — tedy minimální a maximální délku výrazu, kolik různých výrazů jedné délky se bude opakovat a také jak velké skoky budou mezi jednotlivými délkami. Příklad takto vygenerovaného souboru můžete shlédnout ve Výpise 2.

```
(3)*(0)*
(x + ((7)* + u))
(V + ((z)* + (K)*))
(QKP + (D)*)
(X + G)((X)*7)*
(p + (I)*(a)*(o)*)
```

Výpis 2: TXT soubor vygenerovaný s parametry min=5, max=10, opak=1, step=1

Podrobnější popis fungování GUI naleznete v uživatelském manuálu, který se nachází v Příloze A.

4.2 Aplikace z pohledu implementace

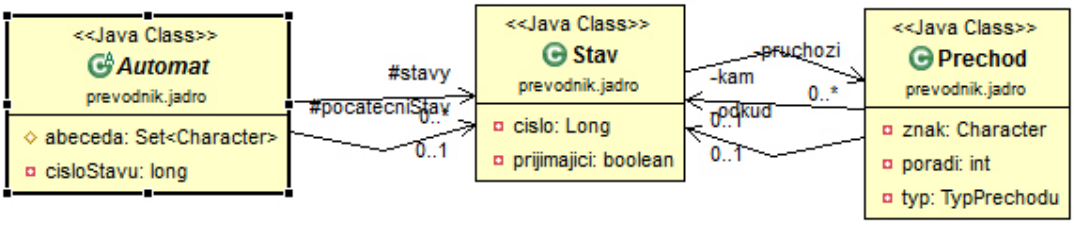


Obrázek 20: Třídní diagram hiearchie tříd tvořící základ pro reprezentaci syntaktického stromu, zde konkrétně ukázka vztahu dědičnosti pro třídu reprezentující uzel typu zřetězení.

Jednotlivé třídy jsou logicky roztříděny do packages:

- **dka** — Obsahuje třídy, prostřednictvím kterých se buďto vytváří DKA, nebo jsou zde třídy, které obsahují algoritmy pro převod, jejichž výstupem je právě DKA.
- **dot** — Shromažďuje třídy obsluhující generování vstupu ve formátu DOT pro GraphViz.

- **generator** — Zde se nacházejí algoritmy, které slouží pro generování náhodných výrazů.
- **jadro** — Jedná se o package, kde jsou shromážděny třídy potřebné pro tvorbu automatu (automat samotný, třídy pro stav a přechod).
- **nka** — Zahrnuje třídy, prostřednictvím kterých se buďto vytváří NKA, nebo jsou zde třídy, které obsahují algoritmy pro převod, jejichž výstupem je právě NKA.
- **rv** — Zde se nacházejí třídy pro reprezentaci regulárního výrazu. A to jak v podobě stringu, tak v podobě stromu. Jsou zde tedy obsaženy třídy pro jednotlivé druhy uzlů, třída reprezentující abecedu a také parser, který převádí výrazy ze stringu do stromové struktury.
- **ui.swing** — Třídy pro GUI.

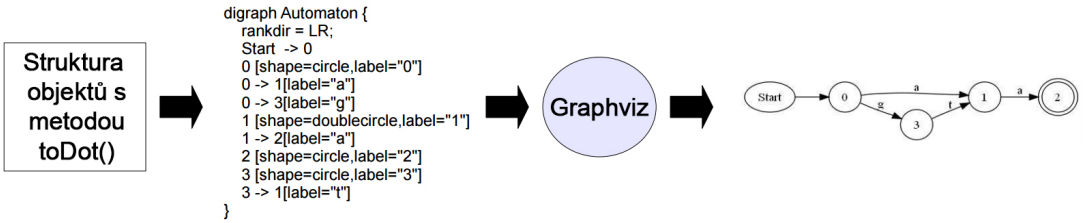


Obrázek 21: Třídní diagram hiearchie tříd tvořící základ pro reprezentaci obecného modelu konečného automatu (metody nejsou z důvodu přehlednosti zobrazeny).

Nejzákladnější třídy jsou třída *Uzel*, která je rodičovskou třídou pro všechny možné druhy uzlu, které se mohou vyskytnout v syntaktickém stromu (například $*, \varepsilon, \emptyset$, symbol, atd. — příklad na Obrázku 20); *Automat*, která je rodičovskou třídou pro všechny druhy automatu, které mohou při převodu vzniknout (ZNKA, NKA, DKA — příklad na Obrázku 21); *Stav*, která reprezentuje stavy automatu a také *Prechod*, která reprezentuje přechody automatu.

Grafická reprezentace stromových struktur a automatů je v implementaci řešena pomocí Graphviz, což je open source grafový vizualizační software. Jeho hlavní devizou je jeho snadná použitelnost — na základě vstupu v textovém formátu automaticky vykreslí požadovaný graf bez jakýchkoliv dalších výpočtů. Konkrétně se jedná o vstup ve formátu DOT (graph description language). Vzhledem k tomuto faktu obsahuje každý element v implementaci aplikace, který budeme vykreslovat (tedy stavy, přechody, uzly), metodu *toDot()*, která vrací řetězec právě ve formátu DOT pro vykreslení elementu v grafu. Postupným skládáním těchto elementů, respektive výstupů jejich metod *toDot()* vznike řetězec ve formátu DOT pro vykreslení celého grafu či stromu.

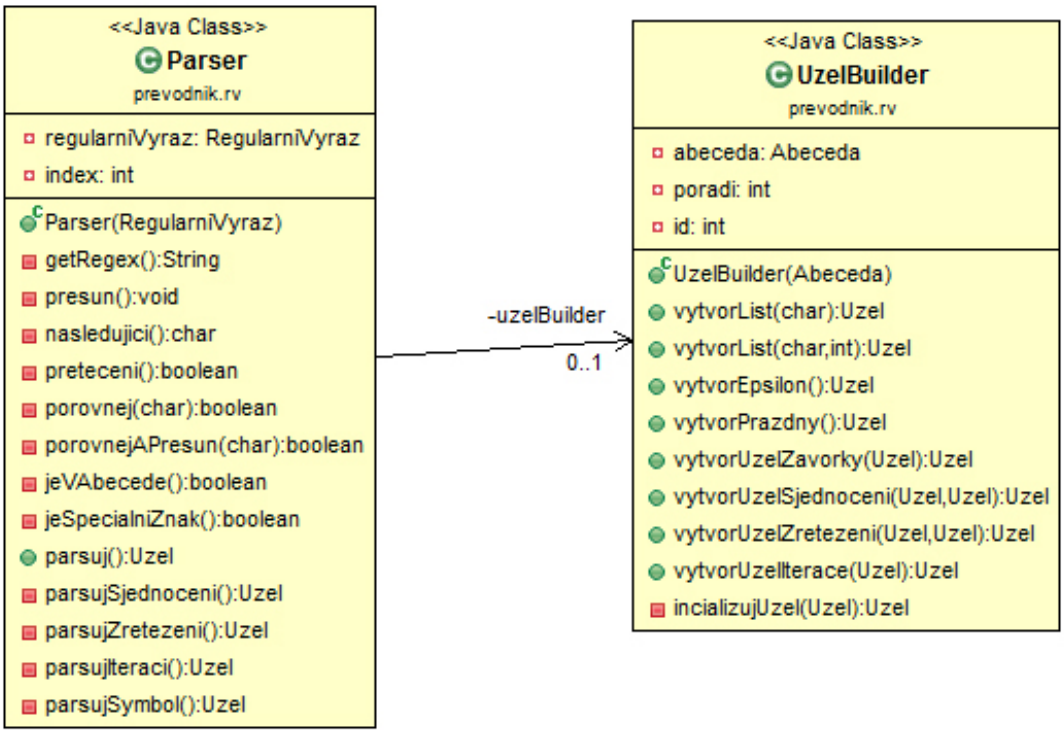
Na výše uvedených třídách stojí veškerá další implementace převodu regulárních výrazů na konečné automaty. Grafická reprezentace syntaktického stomu a konečného automatu je taktéž využívána při každém převodu. Výstupem každého algoritmu je tedy nejen hierarchie objektů



Obrázek 22: Znázornění vygenerování grafické reprezentace datové struktury z hierarchie objektů.

v paměti tvořící jeden celek v podobě konečného automatu, ale taktéž DOT řetězec, který je vstupem pro Graphviz, díky kterému je následně vygenerována grafická podoba automatu (Obrázek 22).

4.2.1 Třídy pro převod regulárního výrazu na automat



Obrázek 23: Parser má k dispozici instanci UzelBuildera, pomocí kterého je sestavována struktura syntaktického stromu.

4.2.1.1 Parsování řetězce Prvním krokem na začátku vyvíjené aplikace je vždy převod regulárního výrazu z textové podoby do podoby syntaktického stromu. K tomu slouží třída *Parser*, ve které parser postupně naráží na operandy ($\emptyset, \varepsilon$ a symboly abecedy) a operace (sjednocení, zřetězení a iterace). Parser na základě těchto operandů a operací vytváří uzly syntaktického

stomu, a to pomocí třídy *UzelBuilder*. Tato třída obsahuje metodu pro každý typ vrcholu, tedy *vytvorPrazdny(), vytvorEpsilon(), vytvorList(), vytvorUzelSjednoceni(levyPotomek, pravyPotomek), vytvorUzelZretezeni(levyPotomek, pravyPotomek) a vytvorUzelIterace(potomek)* (viz Obrázek 23).

Máme tedy regulární výraz převedený do tvaru syntaktického stromu, nyní můžeme přejít k samotnému převodu na konečný automat. Následuje popis toho, jak jsou jednotlivé metody a třídy pro převod regulárního výrazu na konečný automat implementovány. Kompletní dokumentace ve formátu Javadoc se nachází v příloze.

4.2.1.2 Thompsonův algoritmus Každý speciální případ uzlu syntaktického stromu, tedy třídy pro operace a operandy, obsahují metodu *inkrementalniAlg()* (interně je v aplikaci Thompsonův algoritmus pojmenován jako inkrementální), která postupně vytváří zobecněný nedeterministický konečný automat, respektive jeho část odpovídající podvýrazu stomu, jehož kořenem je uzel, ve kterém se právě algoritmus nachází. Pokud tedy v syntaktickém stromu algoritmus narazí například na operand sjednocení, zavolá rekuzrivně metodu *inkrementalniAlg()* na levého i pravého potomka, přičemž oba potomky vytvořené podautomaty následně spojí ε-přechody způsobem popsaným v teoretické části (tedy nový počáteční a koncový stav, ε-přechody z nového počátečního stavu do původních, stejně tak ε-přechody z původních koncových stavů do nového).

4.2.1.3 Berry/Sethi deriváty Pro převod pomocí derivátů slouží samostatná třída *DerivativesPrevod*. Na jejím samotném začátku se vytvoří lokální kopie syntaktického stromu odpovídajícího regulárního výrazu. Pak již probíhá samotné vytváření derivátů pomocí pravidel definovaných v Kapitole 3.2. Používají se metody *derive(uzel, znak)* (výsledkem metody je výsledek operace levý kvocient podle vstupního znaku), *delta(uzel)* a *formalizuj(uzel)*. K porovnání dvou stromů, abychom věděli, zda vytvářet nový stav, nebo pouze vést přechod do již existujícího, slouží metoda *porovnejStromy(strom1, strom2)*. Před spuštěním algoritmu pro porovnání dvou stromů je na každý ze dvou porovnávaných syntaktických stromů aplikována metoda *upravStromHledani(uzel)*, která zajistí změnu původní struktury syntaktického stromu do tvaru vhodného pro porovnávání dvou syntaktických stromů, tzn. do tvaru, kde vnitřní uzel může mít více než jednoho potomka (s ohledem na asociativitu, komutativitu a idempotenci).

Pokud se po provedení operace levý kvocient ukáže, že v automatu ještě neexistuje stav odpovídající nově vzniklému regulárnímu výrazu, je tento stav zařazen na frontu nezpracovaných stavů. Jakmile je stav z této fronty zpracován (tedy je na něj aplikována operace levého kvocientu podle všech znaků abecedy Σ), je z fronty odstraněn. Algoritmus běží dokud je tato fronta neprázdná.

Mimo výše uvedené operace je zde potřeba ještě uvést obecně platná pravidla pro formalizaci derivátů, která je vždy nutné aplikovat na nově vzniklý derivát a to z důvodu, abychom pracovali vždy s co nejjednodušším výrazem.

Pro operaci $formalizuj(E)$ platí:

- $\emptyset + E \equiv E + \emptyset \equiv E$
- $\emptyset \cdot E \equiv E \cdot \emptyset \equiv \emptyset$
- $\varepsilon \cdot E \equiv E \cdot \varepsilon \equiv E$

Generování nových stavů je tedy objasněno. Jak ale vyplývá z výše uvedeného postupu, po vygenerování derivátu je nutné zjistit, zda již neexistuje stav reprezentující stejný regulární výraz, jako právě nově získaný derivát. Jinými slovy je třeba zjistit, zda budeme v automatu vytvářet nový stav, nebo zda pouze vytvoříme přechod do již existujícího stavu.

Vytvořit operaci porovnávání dvou syntaktických stromů regulárních výrazů je komplikované. Mohli bychom jednoduše zjišťovat, zda $E = F$. Bude se jednat o jednoduchý úkol, ale je zde reálné riziko, že generování nových stavů automatu se nemusí nikdy zastavit, protože například pro výrazy $E = (a + b)$ a $F = (b + a)$ rovnost platit nebude, ačkoliv jejich automaty přijímají stejný jazyk. Nabízí se tedy myšlenka porovnávat, zda $L(E) = L(F)$, kde je jistota, že se generování stavů konečného automatu zastaví.

Pro účely této práce je využíván jakýsi hybrid mezi dvěma výše uvedenými přístupy. Porovnáváme, zda $E = F$, ale oba syntaktické stromy jsou nejprve upraveny podle pravidel asociativity, komutativity a idempotence.

To se sebou nutně nese úpravu samotné datové struktury syntatický strom. Nyní zde již nemusejí být pro každý nelistový uzel pouze dva pomoci, ale může jich být i více. Tato úprava je řešena v metodě *upravaStromu()*.

Pro operaci $upravaStromu(E)$ platí:

- $E_1 + (E_2 + E_3) \equiv (E_1 + E_2) + E_3$
- $E_1 + E_2 \equiv E_2 + E_1$
- $E_1 + E_1 \equiv E_1$
- $E_1 \cdot (E_2 \cdot E_3) \equiv (E_1 \cdot E_2) \cdot E_3$

Alrogitmus pro konstrukci deterministického konečného automatu D:

1. Stavy automatu D jsou všechny od sebe různé deriváty $w\backslash E$.
2. Vytvoř přechod pro symbol a ze stavu p do stavu q právě tehdy, když p je pro derivát $w\backslash E$, pro nějaké w, a q je pro $(wa)\backslash E$.
3. Stav pro E je počáteční stav. Stav je přijímající právě tehdy, když je pro derivát $w\backslash E$, pro nějaké w a $\delta(w\backslash E) = \varepsilon$, tj. $\emptyset \in L(w\backslash E)$

4.2.1.4 Berry/Sethi odlišné znaky

V tomto algoritmu je replikován kód z předchozího algoritmu, nepoužívá se zde metoda pro porovnávání dvou syntaktických stromů a tím pádem ani metoda pro jejich úpravu do tvaru vhodného pro porovnávání. Hlavním rozdílem zde je, že každý symbol abecedy je odlišný, protože každý symbol je označen číselným indexem. To znamená, že při generování derivátu si musíme dávat pozor na to, abychom při porovnávání listových uzlů nebrali ohled pouze na ohodnocení symbolem, ale také na označení číselným indexem.

Vše se odehrává ve třídě *DistinctPrevod*. Vygenerování stavů automatu je zde jednoduché, pro každý unikátní znak abecedy odlišený hodnotou indexu vytvoříme jeden stav a k tomu jeden stav počáteční. Pro vyenerování přechodů použijeme znovu metodu *derive(uzel, znak)* (byť mírně upravenou kvůli indexování operandů) pro každý výraz odpovídající jednotlivým stavům automatu. Pokud je výsledkem operace levého kvocientu cokoliv jiného, než $\emptyset$, pak vedeme přechod z aktuálně zpracováváného stavu do stavu odpovídajícímu znaku, který byl dosazen do parametru metody *derive*.

Algoritmus konstrukce nedeterministického konečného automatu M' z očíslovaného regulárního výrazu E':

1. M' má stav pro každého následníka pro každý očíslovaný symbol v E'.
2. Vytvoř přechod pro symbol a ze stavu p do stavu pro následníka pro a právě tehdy, když p je předchůdce pro nějakého následníka C a C může generovat řetězec, který začíná na a.
3. Stav pro E' je počáteční stav. Stav je přijímající právě tehdy, když je následníkem C a $\delta(C) = \varepsilon$.

4.2.1.5 Berry/Sethi rychlý algoritmus

Zde již je úplně vypuštěno generování derivátů pomocí levého kvocientu, což celý proces zjednodušuje a zrychluje. Pracujeme ve třídě *FastPrevod*. Ze všeho nejdříve si zřetězíme regulární výraz E s koncovým speciálním znakem !. Následně si rekurzivně projdeme celý strom a pro každý uzel si vygenerujeme množiny *first* a *follow*.

Automatu znovu vytvoříme stavy podle operandů výrazu, respektive podle listových položek syntaktického stromu. Přidáme nový počáteční stav. Přechody zde nyní tvoříme podle prvků v množinách *follow*. Přechody z počátečního stavu vedeme do prvků obsažených v množině *first(E!)*. Pokud se nám v některé z těchto množin objeví přechod do listového uzlu s hodnotou operandu !, nevedeme přechod, ale zdrojový stav označíme jako koncový.

Algoritmus pro „rychlou“ konstrukci nedeterministického konečného automatu M' z očíslovaného regulárního výrazu E':

1. M' má stav pro každý očíslovaný symbol v E' plus jeden počáteční stav.
2. Vytvoř přechod pro symbol a z počátečního stavu do stavu pro a_i právě tehdy, když $a_i \in first(E')$, vytvoř přechod ze stavu pro b_j do stavu pro a_i právě tehdy, když $a_i \in follow_{E'!}(b_j)$.

3. Stav pro a_i je přijímající stav právě tehdy, když ! $\in follow_{E'!}(a_i)$. Počáteční stav je přijímající právě tehdy, když ! $\in first(E')$

4.2.1.6 Bruggemann-Klein algoritmus Modifikace předchozího algoritmu. Ve třídě *PositionPrevod* nyní generujeme množiny *follow* ve vztahu k pozicím, přechody pak sestavujeme z množiny *follow* pro celý výraz E. Novinkou je generování množiny *last*.

Algoritmus konstrukce nedeterministického konečného automatu M' z očíslovaného regulárního výrazu E':

1. M' má stav pro každý očíslovaný symbol v E' plus jeden počáteční stav.

2. Vytvoř přechod pro symbol a z počátečního stavu do stavu pro a_i právě tehdy, když $a_i \in first(E')$, vytvoř přechod ze stavu pro a_i do stavu pro b_j právě tehdy, když $b_j \in follow(E, a_i)$.

3. Stav pro a_i je přijímající stav právě tehdy, když $a_i \in last(E, a_i)$. Počáteční stav je přijímající právě tehdy, když $\varepsilon \in L(E)$.

4.2.1.7 Optimalizovaný Brüggemann-Klein algoritmus s nullable proměnnou Tento algoritmus se nachází ve třídě *PositionPrevodNullable*. Je zde zavedena nová proměnná *nullable*, která určuje, zda daný podstrom generuje prázdné slovo, či nikoliv. Vygenerované množiny *first*, *last*, *follow* a proměnná *nullable* se nyní navíc ukládají do globální proměnné *globalniPromenna*, přičemž strom je procházen (a potřebné množiny generovány) od listů ke kořeni. Časová složitost zde ještě ale není příliš optimální — při určitých operacích *sjednocení* dvou množin musíme ověřovat duplicitu prvků se složitostí $O(n)$. Z vlastností algoritmu sice vyplývá, že některé množiny lze sloučit sjednocením bez ověřování duplicit, protože jsou disjunktní, v některých případech ale toto zaručeno není. I v tomto algoritmu je tedy na všech místech, kde k takovémuto sjednocování dochází, ověřována duplicita prvků ve výsledné množině vzniklé operací sjednocení.

4.2.1.8 Optimalizovaný Bruggemann-Klein algoritmus s nullable proměnnou a vstupem ve star normal form Implementace tohoto algoritmu vychází z faktu, že pokud kompletně eliminujeme při operacích sjednocení nutnost ověřovat duplicity, zmenšíme časovou složitost. Jeden druh, který je popsaný v teoretické části, nám vypuštění tohoto ověření brání. Zbavíme se ho tak, že celý syntaktický strom převedeme do SNF. Vše se odehrává ve třídě *PositionPrevodNullableSNF*. Jako v každém předchozím případě je lokálně nakopírován syntatický strom, se kterým pracujeme. Pomocí metody *SNF()* je strom převeden do SNF a to jedním průchodem, přesně jak je definováno v teoretické části (tzv. operace s „plným“ a „prázdným“ kolečkem). S takto upraveným stromem si pak již můžeme dovolit vypustit ověřování duplicit při sjednocování množin uvnitř uzlů stromu. Algoritmus je tedy totožný s předchozím, ale jeho časová složitost je menší, právě díky zajištěnému vstupu, který umožňuje vynechat operaci ověřování duplicit.

4.2.1.9 CFS algoritmus V kapitole 3.8 je uvedeno jakým způsobem rozdělujeme strom na podstromy a jak sestavujeme automat podle C množin vzniklých dekompozicí. Nyní bude upřesněno jakým konkrétním algoritmem je tato dekompozice prováděna.

Ze všeho nejdříve potřebujeme funkci *next*. Ta nám určuje následníka ve stromu (při zřetězení ukazuje na „pravého souseda“, při iteraci na sebe sama).

$$next(F) = \begin{cases} F & \text{když F je potomkem } F^* \text{ v } t_E, \\ G & \text{když F je potomkem } FG \text{ v } t_E, \\ \bullet & \text{ve všech ostatních případech.} \end{cases}$$

Pak je třeba ještě definovat $first(\bullet) = \oslash$.

Jakmile se při rekurzivním procházení stromu dostaneme do fáze vynořování směrem ke kořeni a zjistíme, že jsme v místě, kde došlo k rozdělení na podstomy t_1 a t_2, začneme přidávat prvky do globální množiny nazvané *dec*. Můžeme si to představit jako výsledek imaginární funkce *dec(x, t_E)*, kde t_E je strom pro výraz E a $x \in pos(E)$. Při tom se řídíme těmito pravidly:

- Pokud jsme v t a v t je jediná pozice:

$$\text{dec(x, t)} = \begin{cases} \{\{x\}\} & \text{když v t existuje uzel F jinde než v kořeni takový, že} \\ & x \in last(F) \cap first(next(F)), \\ \{\oslash\} & \text{ve všech ostatních případech.} \end{cases}$$

- Pokud jsme v t, $|t| > 1$, t_1 je podstrom pod libovolným uzlem F_1 a t_2 je zbytek z t po odstranění t_1:

  Pokud $x \in pos(t_1)$:

$$dec(x,t) = \begin{cases} dec(x,t_1) \cup \{C_1\} & \text{když } x \in last(F_1), \\ dec(x,t_1) & \text{ve všech ostatních případech,} \end{cases}$$

  přičemž

$$C_1 = pos(t) \cap \bigcup_{\substack{F \prec G \preceq F_1 \\ last(G) \cap pos(F_1) = last(F_1)}} first(next(G)).$$

  Pokud $x \in pos(t_2)$:

$$dec(x,t) = \begin{cases} dec(x,t_2) \cup \{C_2\} & \text{když } first(F_1) \subseteq follow(E,x), \\ dec(x,t_2) & \text{ve všech ostatních případech,} \end{cases}$$

  přičemž

$$C_2 = pos(t) \cap first(F_1).$$

Alrogitmus pro vytvoření dekompozice *follow* množin:

1. Když $|t| = 1$, proveď dekompozici pro jednu pozici.

2. Když $|t| > 1$, proveď tyto kroky:

   Začni od kořene t hledat směrem dolů uzel F_1 takový, že $|t|/3 \leq |t_1| \leq 2|t|/3$, kde t_1 je podstrom t pod uzlem F_1.

   Nyní t_2 vznikne odstraněním t_1 z t.

3. Rekurzivně spusť algoritmus na t_1 a t_2.

4. Takto určíme *dec(x, t) pro každé* $x \in pos(t)$.

Tento algoritmus je naimplementován ve třídě *CFSPrevod1*.

4.2.2 Implementace dalších potřebných algoritmů

NKAdoDKA a *DKAMinimalizace* jsou třídy, jak název napovídá, určené k převádění vygenerovaných konečných automatů do jednotné podoby. Jejich funkčnost odpovídá algoritmům popsaným v teoretické části této práce, podrobný soupis struktury těchto tříd je uveden v dokumentaci.

4.2.2.1 Generátor regulárních výrazů V práci jsou dvě třídy pro generování regulárních výrazů.

První z nich pracuje čistě s textovými řetězci. Jedná se o třídu *GeneratorRozdeleni*. Třída pracuje tak, že rozděluje sekvenci terminálních znaků na regularní operace s nimi, čímž vznikne správně uzávorkovaný regulární výraz. Na vstup je přiveden požadovaný počet terminálních znaků (symbolů), které chceme, aby výraz obsahoval, k tomu je ještě jako parametr předána abeceda, nad kterou je výraz generován. Nejprve se vygeneruje náhodná sekvence znaků zadané délky z požadované abecedy. Například při zadané délce 5 a abecedě {a, b, c} se vygeneruje sekvence *abcab*. Poté se opět náhodně vyberou podvýrazy a nad tyto podvýrazy se přiřadí náhodné operace $(*, \cdot, +)$. Třída pracuje rekurzivně, takže z podvýrazu se mohou stát další podvýrazy s dalšími přiřazenými operacemi. Výsledkem může být například regulární výraz $a^*(b(c + a^*)b)$. Při generování tímto způsobem jsme tedy omezeni na zadaný počet pozic, ale samotná délka výrazu může „nabobtnat“. Tato třída je však použita pouze pro generování krátkých ilustračních výrazů v GUI pro grafický výstup (tlačítko „random RV“ u políčka pro manuální převod jediného výrazu) a tento fakt nám tedy nevadí.

Druhým přístupem ke generování regulárních výrazů je vytváření přímo stromové struktury reprezentující tento výraz. Děje se tak ve třídě *GeneratorUzly*. I zde je jako parametr předána délka výrazu, do této délky jsou však nyní započítány i operace. Jinými slovy, výrazy se stejně zadanou délkou budou mít vždy stejný počet uzlů. K tomu je samozřejmě předána taktéž abeceda, se kterou pracujeme. Třída tedy začne postupně vytvářet stromovou strukturu, přičemž si v mezipaměti ukládá kolik znaků ještě zbývá do maximální zadané délky výrazu. V momentě kdy začne tzv. „docházet místo“, tedy výraz začne atakovat hranici maximální délky, je nutné počítat s tím, že některé operace musejí mít dva potomky $(\cdot, +)$ a některé potomka jednoho $(*)$. Je zde tedy ošetřeno, že jako poslední prvek vložený do stromové struktury musí být jednoznačně

pouze znak. Pokud zbývají poslední dvě pozice, je zřejmé, že vložena může být pouze operace $*$ s potomkem, kterým může být opět pouze znak. Operace $(\cdot, +)$ tedy mohou být vkládány pouze v případě, že zbývají 3 a více míst. Operace a znaky jsou i zde vkládány zcela náhodně. Uživatel si však v GUI může zadat minimální a maximální délku, včetně abecedy, ze které bude výraz vygenerován. Tento způsob generování je použit pro vytváření TXT souborů obsahující větší množství výrazů větších délek (řádově tisíce), které jsou určeny pro hromadný převod, při kterém jsou mezi sebou algoritmy pro převádění porovnávány. V případě, že se vygenerovaný výraz neexportuje do TXT souboru, je možné ihned předat jeho kořen algoritmu pro převod, odpadá zde tedy nutnost parsovat výraz z textové podoby do stromové struktury.

4.2.3 Kompilace a spuštění

Aplikace je kompilována prostřednictvím Ant. Soubor build.xml je součástí zdrojových kódů aplikace. Při kompilaci je taktéž vytvořena dokumentace Javadoc. Výsledkem kompilace je soubor *prevodnik-regexp.jar*. Spuštění aplikace je vyvoláno skrze **java -jar prevodnik-regexp.jar**.

5 Testování naimplementovaných algoritmů

Každý z naimplementovaných algoritmů má vždy stejný vstup, výstupy se ale mohou různit. Jeden z algoritmů umí regulární výazy převádět přímo do DKA, další algoritmy to zvládnou pouze do NKA a jeden pouze do ZNKA. Za konečný výstup tedy budeme považovat co nejmenší a nejúspornější možnou variantu, tedy **detereministický konečný automat v minimálním tvaru**. Pro tento účel aplikace přirozeně obsahuje i implementaci převodu z nedeterministického konečného automatu na detereministický a také třídu pro minimalizaci.

Pro stejný výraz převodem obdržíme vždy ekvivalentní deterministický automat, v minimálním tvaru pak má tento výsledný automat i totožný graf. Každý algoritmus nám tedy pro stejný vtup dává stejný výstup (vyjma CFS algoritmu, který umí snížit počet přechodů). Jelikož každý z těchto algoritmů pracuje jinak, zajímá nás **čas**, za jaký algoritmus převod provede. Z tohoto hlediska tedy algoritmy porovnáme.

Cesta od vstupu k výstupu u každého z algoritmů zahrnuje jinou posloupnost operací, a to konkrétně:

- **Thompsonův algoritmus**

  Regulární výraz → zobecněný nedeterministický konečný automat → detereministický konečný automat → minimalizace.

- **Berry/Sethi derivatives**

  Regulární výraz → detereministický konečný automat → minimalizace.

- **Berry/Sethi distinct symbols + fast algoritmus, Brüggemann-Klein algoritmus + algoritmus s nullable proměnnou, CFS algoritmus**

  Regulární výraz → nedeterministický konečný automat → detereministický konečný automat → minimalizace.

- **Optimalizovaný Brüggemann-Klein algoritmus s nullable proměnnou a vstupem v SNF**

  Regulární výraz → převod do SNF → nedeterministický konečný automat → detereministický konečný automat → minimalizace.

5.1 Předpoklady

- Předpokladem **nejméně efektivního algoritmu** je Berry/Sethi derivatives. Algoritmus sice převádí výraz rovnou do DKA, ale jeho efektivita závisí na nutnosti ověřování rovnosti derivátů. V nejhorším případě může algoritmus pro výraz t o velikosti n dosáhnout složitosti $O(||t||^2 * n^2)$ (bez minimalizace).

- **Nejefektivnějším algoritmem** by naopak měl být CFS algoritmus. Ten vychází z Position algoritmu, který má v základní verzi kubickou složitost. V tomto případě je však složitost $O(n * log(n)^2)$ (bez převodu do DKA a minimalizace). Pokud navíc nastane případ, že se oproti ostatním algoritmům sníží počet přechodů, budou tímto ovlivněny i další operace (převod do DKA a minimalizace).

5.2 Průběh testování

Měření probíhalo PC stanici s Intel Core i5, 8GB RAM a Windows 10 64bit. Porovnávání převodu jednotlivých výrazů je obstaráno prostřednictvím třídy, která jako vstup přijímá TXT soubor obsahující regulární výrazy oddělené řádky.

Tento vstupní soubor byl vygenerován pomocí naimplementovaného generátoru s parametry:

- Minimální délka: 100.
- Maximální délka: 1900.
- Opakování: 10.
- Step: 200.

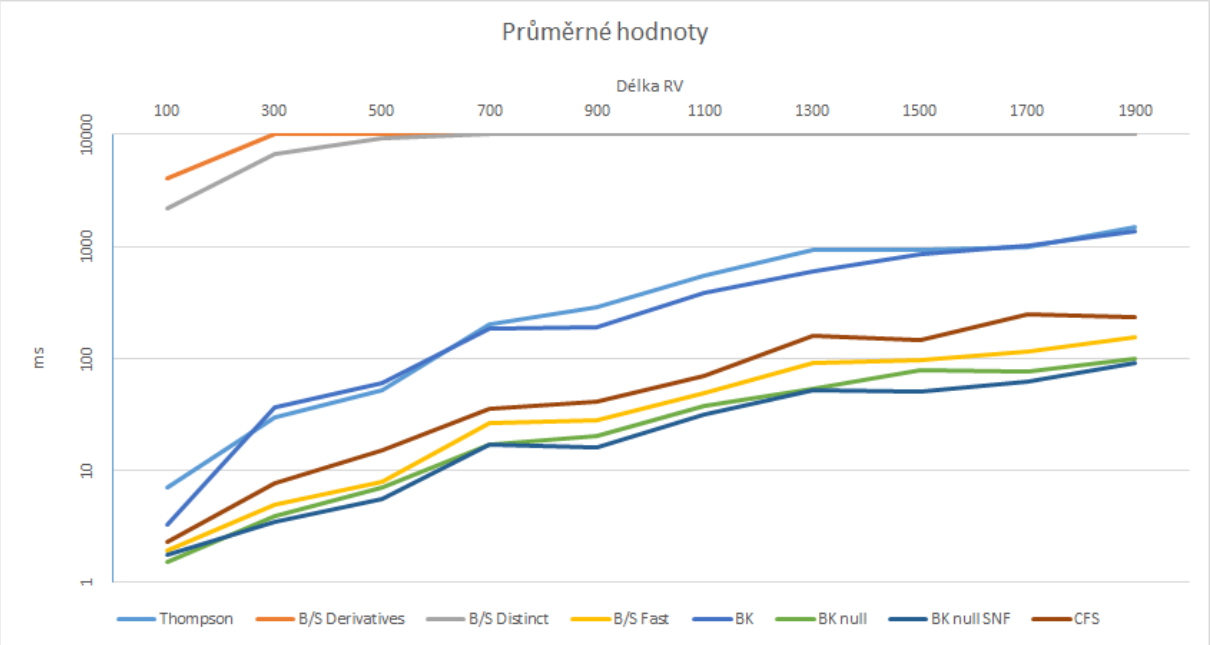
Každý výraz je postupně převeden prostřednictvím každého z algoritmů a při tom je měřen čas. Přesný postup zpracování regulárních výrazů ze zdrojového TXT souboru je tento:

1. Otevři zdrojový TXT soubor.
2. Načti první nezpracovaný řádek s regulárním výrazem.
3. Spusť měření času.
4. Převeď regulární výraz na automat prostřednictvím vybraného algoritmu.
5. Zastav měření času.
6. Ulož naměřené hodnoty v CSV + výsledný automat v XML.
7. Opakuj postup od bodu 3 s dosud nepoužitým algoritmem.
8. Pokud existuje nezpracovaný řádek, opakuj postup od bodu 2.
9. Konec hromadného převodu, zavři TXT soubor.

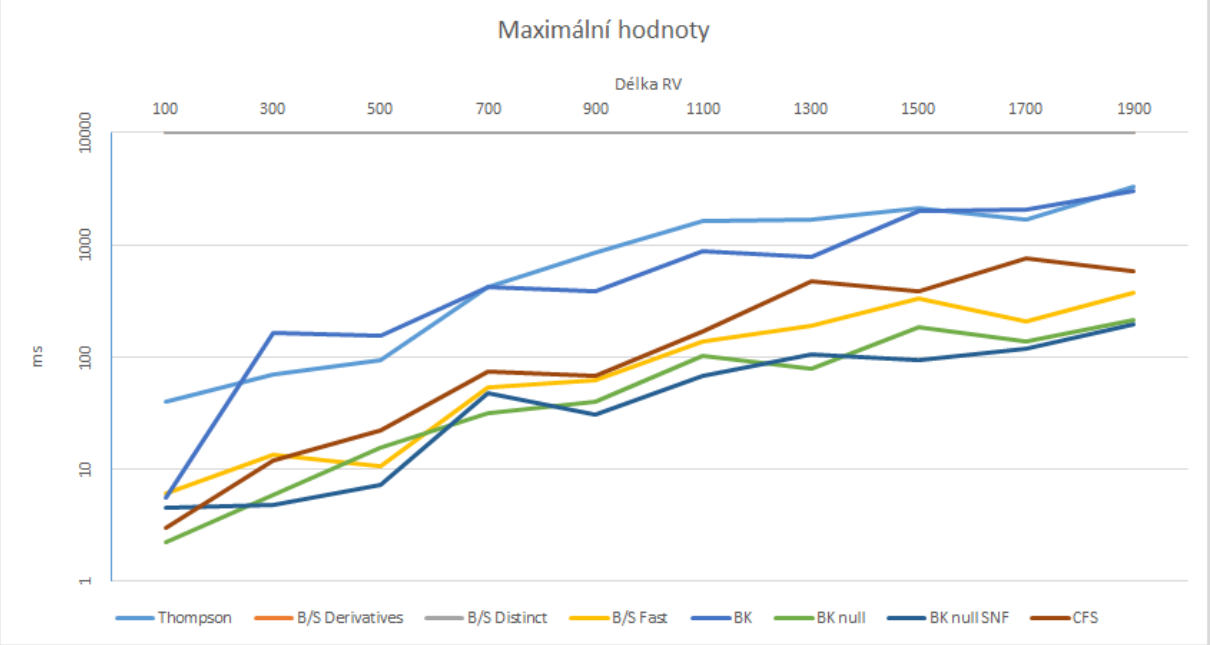
Při vykonávání jednotlivých převodů se bere ohled na uživatelem zadaný časový limit v milisekundách. Tento limit musí být dodržen pro každý mezikrok (samotný převod, převod z NKA do DKA, minimalizace). Pokud je limit u jakéhokoliv z kroků převodu překoročen, pak algoritmus převodu končí a do statistik je zapsána hodnota časového limitu pro všechny následující operace. Díky tomu je pak ve výsledcích testování možné zjistit ve kterém kroku převodu došlo k vypršení časového limitu.

5.3 Výsledky testování

Testování proběhlo nad dvěma abecedami — první o velikosti 1, druhá o velikosti 50 znaků. Zadaný časový limit byl 10 000 ms.

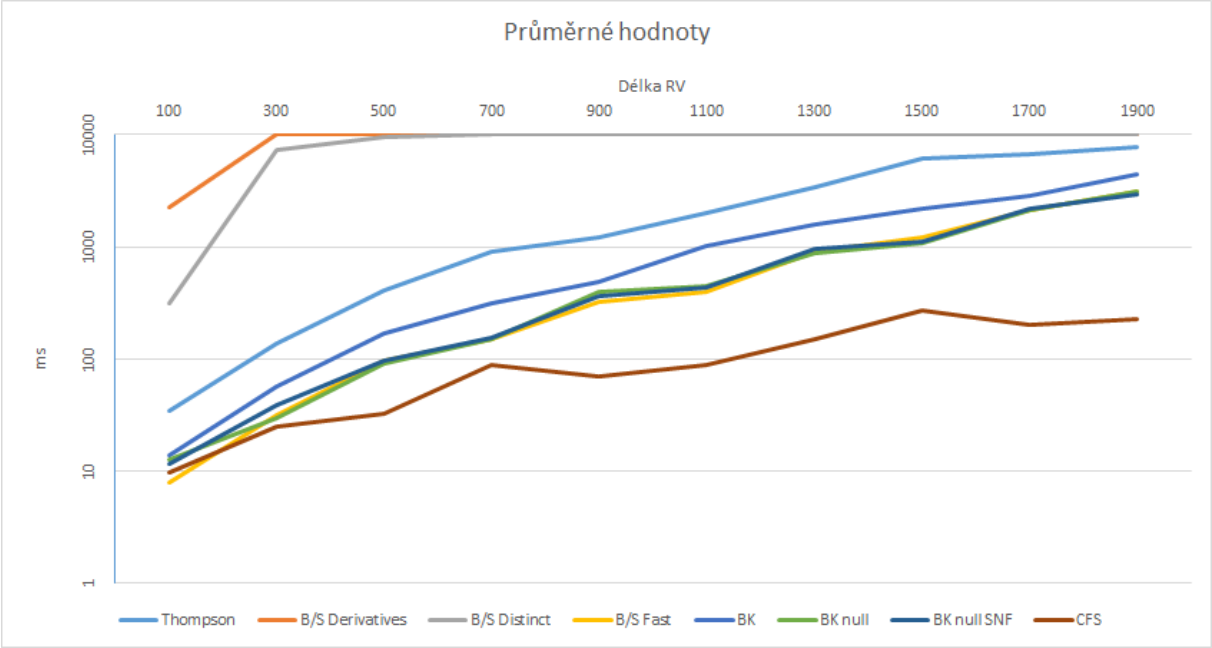


Obrázek 24: Průměrné naměřené časy převodu regulárních výrazů nad abecedou velikosti 1

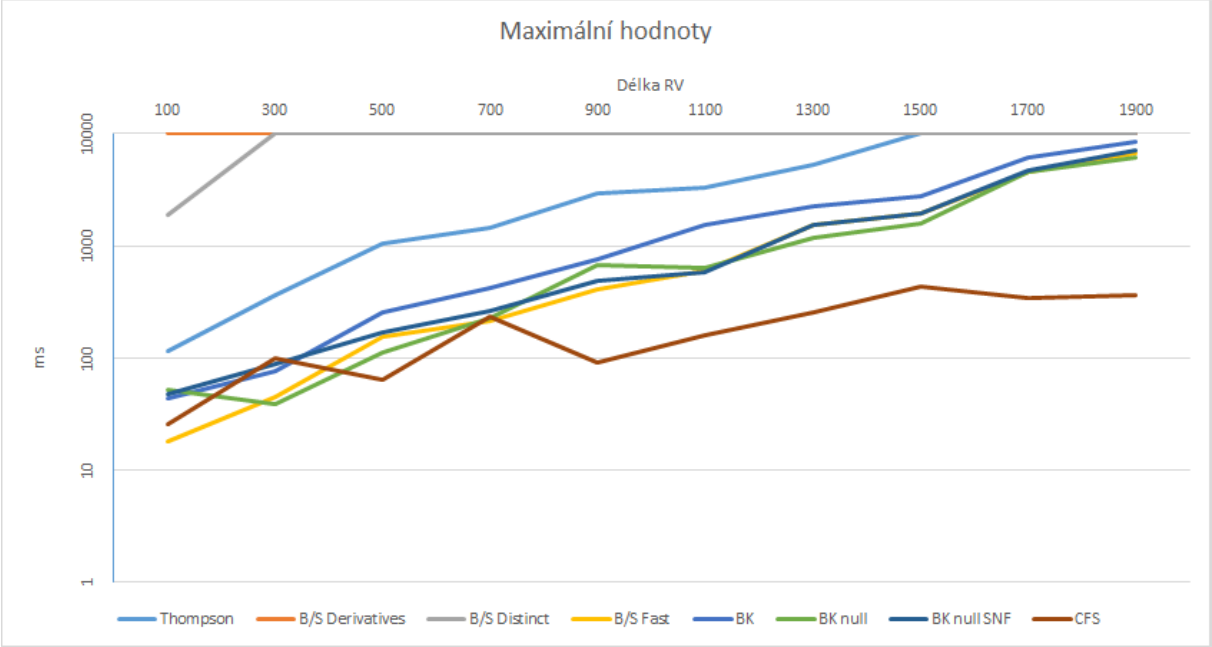


Obrázek 25: Maximální naměřené časy převodu regulárních výrazů nad abecedou velikosti 1

Podrobné výsledky testování jsou k dispozici v Příloze A.



Obrázek 26: Průměrné naměřené časy převodu regulárních výrazů nad abecedou velikosti 50



Obrázek 27: Maximální naměřené časy převodu regulárních výrazů nad abecedou velikosti 50

5.4 Vyhodnocení

Z naměřených výsledků vyplývá, že předpoklady byly z větší části správné v případě, že pracujeme nad abecedou, která obsahuje více znaků. Ve všech případech je nejpomalejším B/S Derivatives algoritmus. Pokud pracujeme nad abecedou s jedním znakem, pak mu zdatně kon-

kuruje B/S Distinct algoritmus. To je způsobeno tím, že tento algoritmus považuje všechny znaky za navzájem odlišné, přestože jsou v abecedě o jednom znaku všechny totožné. Jako nejoptimálnější z časového hlediska se při jednoznakové abecedě jeví B/K SNF algoritmus. Z grafu je také patrný mírnější nárůst času potřebného pro převod v závislosti na délce regulárního výrazu.

S přibývajícím počtem znaků abecedy se však situace mění — předpoklady nejefektivnějšího v takovém případě plní CFS algoritmus, který jako jediný zužitkuje menší velikost NKA vzniklého z RV, díky čemuž je pak ušetřen čas při operacích, které provádějí převod do DKA a minimalizaci. Nárůst času potřebného na převod výrazu s ohledem na jeho délku je zde progresivnější než v jednoznakové abecedě.

6 Future work

Existuje ještě celá řada dalších algoritmů, které by bylo možné použít pro převod regulárního výrazu na konečný automat. Bylo by zajímavé je porovnat s dosavadní implementací a zjistit, zda nějaký dosud popsaný algoritmus pro převod výrazu přímo do determinisitického konečného automatu dokáže tento převod realizovat v čase alespoň vzdáleně se přibližující časům algoritmů, které výraz převádějí do nedeterministického konečného automatu.

Co se týče implementace, byla pro její realizaci zvolena Java. Je zřejmé, že v tomto jazyce nebudeme dosahovat oslnivých výsledků při testování těchto algoritmů. Bylo by vhodné prozkoumat, zda by bylo možné dosáhnout lepších výsledků, kdyby implementace byla realizována v nějakém z „nižších“ programovacích jazyků. Taktéž by se dalo zjistit, zda by některý z algoritmů prezentovaných v této práci mohl mít při implementaci v jiném jazyce jiný výkon ve vztahu k ostatním zde prezentovaným algoritmům.

Co se týče nejvýkonnějšího CFS algoritmu z této práce, bylo by vhodné doimplmentovat i další jeho varianty popsané v článku Translating regular expressions into small ε-free nondeterministic finite automata [4].

Prostor je zde také ponechán v oblasti další optimalizace nejen času, ale zejména paměťové náročnosti a velikosti vzniklých automatů při provádění jednotlivých algoritmů.

7 Závěr

V práci jsem nejprve vytvořil parser, který z regulárního výrazu v textové podobě vytvoří regulární výraz reprezentovaný syntaktickým stromem.

Dále jsem naimplementoval osm různých algoritmů, které mají jako výstup buď zobecněný nedeterministický konečný automat, nedeterministický konečný automat nebo přímo deterministický konečný automat. Do práce byly zavedeny operace převodu (Z)NKA na DKA, minimalizace a převodu na normovaný tvar.

Všechny tyto algoritmy jsem naimplementoval v Javě, přičemž implementace odpovídá teorii těchto algoritmů popsané v této práci. Jedná se zejména o práci a nadefinované operace s množinami. Automaty jsou v drtivé většině případů vytvářeny pomocí rekurzivního procházení syntaktického stromu. Vytvořený automat pak lze po převedení na deterministickou verzi a provedení minimalizace exportovat ve formátu XML.

Naimplementoval jsem generátor regulárních výrazů a také hromadný převod výrazů. Aplikaci lze díky tomu nejprve vygenerovat vstupní soubor s více výrazy a ten následně zadat na vstup hromadného převodu výrazů na automaty. Aplikace má také GUI.

Při testování jsem naimplementované algoritmy nechal proběhnout nad dostatečným počtem výrazů s dostatečným počtem opakování. Při tomto průběhu jsem zaznamenal časy, za které se algoritmy vykonaly, tedy za jak dlouho algoritmy převedly regulární výraz na vstupu do výsledného minimalizovaného deterministického konečného automatu. Získaná data jsem vyhodnotil a porovnal s předpoklady.

Při vyhodnocování jsem zjistill, že je výhodnější regulární výraz nejprve převést na nedeterministický automat, poté na deterministický a následně minimalizovat, než výraz převádět rovnou do deterministické verze automatu.

Literatura

[1] BERRY, Gerard; SETHI, Ravi. From regular expressions to deterministic automata. Theoretical computer science, 1986, 48: 117–126.

[2] BRÜGGEMANN-KLEIN, Anne. Regular expressions into finite automata. Theoretical Computer Science, 1993, 120.2: 197–213.

[3] SIPSER, Michael. Introduction to the Theory of Computation. Boston: Thomson Course Technology, 2006.

[4] HROMKOVIČ, Juraj; SEIBERT, Sebastian; WILKE, Thomas. Translating regular expressions into small ε-free nondeterministic finite automata. Journal of Computer and System Sciences, 2001, 62.4: 565–588.

A Příloha na CD

Příloha obsahuje:

- kompletní implementaci aplikace v Javě
- kompletní programátorskou dokumentaci aplikace v Javadoc
- uživatelskou dokumentaci ve formátu HTML
- kompletní naměřené hodnoty při testování + vstupní TXT soubory pro hromadný převod